US012282525B2

US012282525B2

(12) **United States Patent**
Sade et al.

(10) **Patent No.:** US 12,282,525 B2
(45) **Date of Patent:** *Apr. 22, 2025

(54) **SYSTEMS, METHODS, AND APPARATUSES FOR MATRIX OPERATIONS**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Raanan Sade**, Kibutz Sarid (IL); **Simon Rubanovich**, Haifa (IL); **Amit Gradstein**, Binyamina (IL); **Zeev Sperber**, Zichron Yackov (IL); **Alexander Heinecke**, San Jose, CA (US); **Robert Valentine**, Kiryat Tivon (IL); **Mark J. Charney**, Lexington, MA (US); **Bret Toll**, Hillsboro, OR (US); **Jesus Corbal**, King City, OR (US); **Elmoustapha Ould-Ahmed-Vall**, Chandler, AZ (US); **Menachem Adelman**, Modi'in (IL)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **18/386,771**

(22) Filed: **Nov. 3, 2023**

(65) **Prior Publication Data**

US 2024/0143325 A1 May 2, 2024

Related U.S. Application Data

(63) Continuation of application No. 15/859,268, filed on Dec. 29, 2017, now Pat. No. 11,816,483.

(51) **Int. Cl.**
G06F 9/30 (2018.01)
G06F 17/16 (2006.01)

(52) **U.S. Cl.**
CPC *G06F 17/16* (2013.01); *G06F 9/30036* (2013.01); *G06F 9/30038* (2023.08); *G06F 9/30101* (2013.01)

(58) **Field of Classification Search**
CPC ... G06F 9/30036; G06F 9/30101; G06F 17/16
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,967,388 A   10/1990 Tate
5,247,632 A   9/1993 Newman
(Continued)

FOREIGN PATENT DOCUMENTS

CN   102918495 A   2/2013
CN   103377037 A   10/2013
(Continued)

OTHER PUBLICATIONS

Non-Final Office Action, U.S. Appl. No. 16/398,200, Jul. 28, 2020, 16 pages.
(Continued)

Primary Examiner — Cheng Yuan Tseng
(74) *Attorney, Agent, or Firm* — Nicholson DeVos Webster & Elliott LLP

(57) **ABSTRACT**

Embodiments detailed herein relate to matrix (tile) operations. For example, decode circuitry to decode an instruction having fields for an opcode and a memory address, and execution circuitry to execute the decoded instruction to store configuration information about usage of storage for two-dimensional data structures at the memory address.

15 Claims, 32 Drawing Sheets
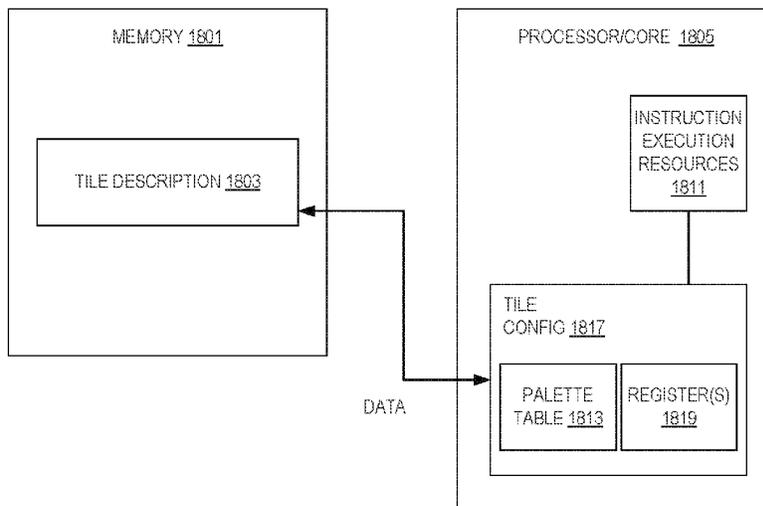
(56) **References Cited**

U.S. PATENT DOCUMENTS

5,475,631	A	12/1995	Parkinson et al.
5,475,822	A	12/1995	Sibigtroth et al.
5,892,962	A	4/1999	Cloutier
6,161,219	A	12/2000	Ramkumar et al.
6,212,112	B1	4/2001	Naura et al.
6,332,186	B1	12/2001	Elwood et al.
6,877,020	B1	4/2005	Bratt et al.
7,003,542	B2	2/2006	Devir
7,209,939	B2	4/2007	Castrapel et al.
7,725,521	B2	5/2010	Chen et al.
7,792,895	B1	9/2010	Juffa et al.
7,844,352	B2	11/2010	Vouzis et al.
7,873,812	B1	1/2011	Mimar
7,912,889	B1	3/2011	Juffa et al.
7,932,910	B2	4/2011	Hansen et al.
8,392,487	B1	3/2013	Mesh et al.
8,417,758	B1	4/2013	Rao et al.
8,924,455	B1	12/2014	Barman et al.
8,984,043	B2	3/2015	Ginzburg et al.
9,442,723	B2	9/2016	Yang et al.
9,557,998	B2	1/2017	Ould-Ahmed-Vall et al.
9,906,359	B2	2/2018	Gueron
9,960,907	B2	5/2018	Gueron
10,535,114	B2	1/2020	Bolz
10,664,287	B2	5/2020	Hughes et al.
10,671,913	B2	6/2020	Liu et al.
10,719,323	B2	7/2020	Baum et al.
10,838,734	B2	11/2020	Hughes et al.
10,853,067	B2	12/2020	Henry et al.
10,860,681	B2	12/2020	Zhang et al.
10,866,786	B2	12/2020	Sade et al.
10,896,043	B2	1/2021	Toll et al.
10,922,077	B2	2/2021	Espig et al.
10,929,503	B2	2/2021	Azizi et al.
10,942,985	B2	3/2021	Espig et al.
10,963,246	B2	3/2021	Heinecke et al.
10,963,256	B2	3/2021	Sade et al.
10,970,076	B2	4/2021	Ould-Ahmed-Vall et al.
10,990,396	B2	4/2021	Toll et al.
11,023,235	B2	6/2021	Sade et al.
11,093,247	B2	8/2021	Sade et al.
11,163,565	B2	11/2021	Valentine et al.
11,294,671	B2	4/2022	Hughes et al.
11,567,765	B2	1/2023	Valentine et al.
11,579,883	B2	2/2023	Hughes et al.
11,669,326	B2	6/2023	Sade et al.
11,809,869	B2	11/2023	Sade et al.
11,816,483	B2 *	11/2023	Sade G06F 9/30101
2002/0198911	A1	12/2002	Blomgren et al.
2003/0126176	A1	7/2003	Devir
2004/0111587	A1	6/2004	Nair et al.
2005/0053012	A1	3/2005	Moyer
2005/0193050	A1	9/2005	Sazegari
2006/0101245	A1	5/2006	Nair et al.
2006/0184837	A1	8/2006	Al-Omari
2006/0190517	A1	8/2006	Guerrero
2007/0186210	A1	8/2007	Hussain et al.
2008/0071851	A1	3/2008	Zohar et al.
2008/0140994	A1	6/2008	Khailany et al.
2008/0208942	A1	8/2008	Won et al.
2009/0043836	A1	2/2009	Dupaquis et al.
2009/0044271	A1	2/2009	Benameur et al.
2009/0113185	A1	4/2009	Hansen et al.
2009/0292758	A1	11/2009	Brokenshire et al.
2009/0300091	A1	12/2009	Brokenshire et al.
2009/0300249	A1	12/2009	Moyer et al.
2010/0180100	A1	7/2010	Lu et al.
2010/0325187	A1	12/2010	Juffa et al.
2012/0079252	A1	3/2012	Sprangle
2012/0113133	A1	5/2012	Shpigelblat
2012/0137074	A1	5/2012	Kim et al.
2012/0254588	A1	10/2012	Adrian et al.
2012/0314774	A1	12/2012	Yang et al.
2013/0013843	A1	1/2013	Radovic et al.
2013/0073599	A1	3/2013	Maloney
2013/0290680	A1	10/2013	Keller et al.
2013/0305020	A1	11/2013	Valentine et al.
2014/0032876	A1	1/2014	Burkart et al.
2014/0095842	A1	4/2014	Caprioli et al.
2014/0149480	A1	5/2014	Catanzaro et al.
2015/0067302	A1	3/2015	Gueron
2015/0081753	A1	3/2015	Iyer et al.
2015/0199266	A1	7/2015	Franchetti et al.
2015/0249468	A1	9/2015	Nikkila et al.
2015/0356054	A1	12/2015	Barak et al.
2016/0063284	A1	3/2016	Tiwari
2016/0179523	A1	6/2016	Ould-Ahmed-Vall et al.
2017/0053012	A1	2/2017	Levy et al.
2017/0053656	A1	2/2017	Moriya et al.
2017/0132008	A1	5/2017	Uliel et al.
2018/0074824	A1	3/2018	Sazegari et al.
2018/0113708	A1	4/2018	Corbal et al.
2019/0042255	A1	2/2019	Sade et al.
2022/0091848	A1	3/2022	Sade et al.
2024/0143325	A1 *	5/2024	Sade G06F 9/30076

FOREIGN PATENT DOCUMENTS

CN	104603766 A	5/2015
EP	2889758 A2	7/2015
KR	10-2011-0079495 A	7/2011
WO	2004/053841 A2	6/2004
WO	2016/003740 A1	1/2016
WO	2016/105727 A1	6/2016
WO	2018/125250 A1	7/2018
WO	2018/174925 A1	9/2018
WO	2018/174935 A1	9/2018

OTHER PUBLICATIONS

Non-Final Office Action, U.S. Appl. No. 17/335,377, Jul. 25, 2022, 5 pages.

Non-Final Office Action, U.S. Appl. No. 17/398,927, Jun. 20, 2022, 7 pages.

Notice of Allowance, U.S. Appl. No. 15/201,442, Dec. 14, 2018, 5 pages.

Notice of Allowance, U.S. Appl. No. 15/858,916, Jun. 15, 2023, 7 pages.

Notice of Allowance, U.S. Appl. No. 15/858,916, Sep. 12, 2023, 2 pages.

Notice of Allowance, U.S. Appl. No. 15/858,932, Apr. 21, 2021, 9 pages.

Notice of Allowance, U.S. Appl. No. 15/858,932, Jun. 2, 2020, 9 pages.

Notice of Allowance, U.S. Appl. No. 15/858,932, May 12, 2021, 7 pages.

Notice of Allowance, U.S. Appl. No. 15/858,932, Sep. 10, 2019, 8 pages.

Notice of Allowance, U.S. Appl. No. 15/858,937, Jul. 11, 2023, 9 pages.

Notice of Allowance, U.S. Appl. No. 15/858,947, Jan. 27, 2021, 9 pages.

Notice of Allowance, U.S. Appl. No. 15/859,268, Jul. 12, 2023, 9 pages.

Notice of Allowance, U.S. Appl. No. 15/859,268, Oct. 13, 2023, 2 pages.

Notice of Allowance, U.S. Appl. No. 15/859,271, Jan. 25, 2023, 13 pages.

Notice of Allowance, U.S. Appl. No. 17/335,377, Feb. 16, 2023, 2 pages.

Notice of Allowance, U.S. Appl. No. 17/335,377, Jan. 10, 2023, 7 pages.

Notice of Allowance, U.S. Appl. No. 17/398,927, Feb. 22, 2023, 5 pages.

Notice of Allowance, U.S. Appl. No. 17/398,927, Jan. 10, 2023, 6 pages.

Notice of Allowance, U.S. Appl. No. 17/398,927, Nov. 17, 2022, 10 pages.

(56) **References Cited**

OTHER PUBLICATIONS

Office Action, DE App. No. 102018010465.7, Jun. 8, 2022, 28 pages of Original Document Only.

Office Action, DE App. No. 102018010565.3, Nov. 23, 2023, 10 pages of Original Document Only.

Office Action, DE App. No. 102018125971.9, Nov. 27, 2023, 12 pages of Original Document Only.

Okuyama et al., "Design of a Coarse-Grained Processing Element for Matrix Multiplication on FPGA", 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, 2014, pp. 237-241.

Pedram et al., "Co-Design Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures", IEEE Transactions on Computers, vol. 61, No. 12, Dec. 2012, pp. 1724-1736.

Pedram et al., "On the Efficiency of Register File versus Broadcast Interconnect for Collective Communications in Data-Parallel Hardware Accelerators", 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing, Oct. 2012, pp. 19-26.

Takahashi et al., "Design and Power Performance Evaluation of On-Chip Memory Processor with Arithmetic Accelerators", 2008 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, Jan. 23, 2008, pp. 51-57.

Wikipedia, "Basic Linear Algebra Subprograms", In: Wikipedia, the free encyclopedia, Available Online at <https://en.wikipedia.org/w/index.php?title=Basic_Linear_Algebra_Subprograms&oldid=S14369141>, Dec. 8, 2017, 10 pages.

Wikipedia, "Communication-avoiding algorithm", In: Wikipedia, the free encyclopedia, Available Online at <https://en.wikipedia.org/w/index.php?title=Communication-avoiding_algorithm&oldid=701848984>, Jan. 26, 2016, 4 pages.

Wikipedia, "Matrix multiplication algorithm", In: Wikipedia, the free encyclopedia, Available Online at <https://en.wikipedia.org/w/index.php?title=Matrix_multiplication_algorithm&oldid=808248148>, Nov. 1, 2017, 10 pages.

Wikipedia, "Multiply-accumulate operation", In: Wikipedia, the free encyclopedia, Available Online at <https://en.wikipedia.org/w/index.php?title=Multiply%E2%80%93accumulate_operation&oldid=784090047>, Jun. 6, 2017, 4 pages.

Wikipedia, "Systolic array", In: Wikipedia, the free encyclopedia, Available Online at <https://en.wikipedia.org/w/index.php?title=Systolic_array&oldid=771564406>, Mar. 22, 2017, 5 pages.

Advisory Action, U.S. Appl. No. 15/859,271, Sep. 20, 2022, 4 pages.

Advisory Action, U.S. Appl. No. 15/858,916, May 8, 2023, 5 pages.

Advisory Action, U.S. Appl. No. 15/858,937, May 18, 2020, 3 pages.

Advisory Action, U.S. Appl. No. 15/859,271, Jul. 18, 2022, 3 pages.

Cascaval et al., "A taxonomy of accelerator architectures and their programming models", In: IBM Journal of Research and Development, vol. 54, No. 5, Sep./Oct. 2010, pp. 5:1-5:10.

Corrected Notice of Allowability, U.S. Appl. No. 15/201,442, Jan. 22, 2019, 5 pages.

Corrected Notice of Allowability, U.S. Appl. No. 15/201,442, Mar. 11, 2019, 2 pages.

Eiron et al., "Matrix Multiplication: A Case Study of Algorithm Engineering", 2nd Workshop on Algorithm Engineering (WAE98), Aug. 20-22, 1998, pp. 98-109.

Final Office Action, U.S. Appl. No. 15/858,916, Mar. 17, 2020, 10 pages.

Final Office Action, U.S. Appl. No. 15/858,916, Nov. 14, 2022, 10 pages.

Final Office Action, U.S. Appl. No. 15/858,916, Sep. 15, 2021, 18 pages.

Final Office Action, U.S. Appl. No. 15/858,937, Feb. 10, 2020, 8 pages.

Final Office Action, U.S. Appl. No. 15/858,937, Sep. 3, 2021, 19 pages.

Final Office Action, U.S. Appl. No. 15/858,947, Oct. 10, 2019, 6 pages.

Final Office Action, U.S. Appl. No. 15/859,268, Jul. 20, 2022, 17 pages.

Final Office Action, U.S. Appl. No. 15/859,268, Feb. 23, 2021, 14 pages.

Final Office Action, U.S. Appl. No. 15/859,268, Sep. 16, 2019, 9 pages.

Final Office Action, U.S. Appl. No. 15/859,271, Aug. 20, 2021, 21 pages.

Final Office Action, U.S. Appl. No. 15/859,271, Jun. 21, 2022, 22 pages.

Final Office Action, U.S. Appl. No. 15/859,271, Mar. 9, 2020, 13 pages.

International Preliminary Report on Patentability, PCT App. No. PCT/US2017/036038, Jan. 17, 2019, 14 pages.

International Preliminary Report on Patentability, PCT App. No. PCT/US2017/040546, Oct. 3, 2019, 10 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/036038, Sep. 5, 2017 , 15 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/040534, Jan. 3, 2018, 11 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/040536, Dec. 20, 2017, 11 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/040537, Dec. 20, 2017, 11 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/040540, Jan. 3, 2018, 14 pages.

International Search Report and Written Opinion, PCT App. No. PCT/US2017/040546, Jan. 24, 2018, 15 pages.

Joerg-Henrik Heine, "RDocumentation—Pair—Rasch Item Parameter", RDocumentation, Version 0.3.1,2015, https://www.rdocumentation.org/packages/pairwise/versions/0.3.1/topics/pair (Year: 2015).

Lahr, David L., "Distractions: Timing Matrix Multiplication in SciDB and Setting the No. of Worker Instances in SciDB and Running Matrix Multiplication Piecemeal", Available Online at <http://dllahr.blogspot.com/2012/11/timing-matrix-multiplication-in-scidb.html>, Nov. 13, 2012, 8 pages.

Merchant et al., "Accelerating BLAS on Custom Architecture through Algorithm-Architecture Co-design", arXiv: 1610.06385v5 [cs.AR], Nov. 27, 2016, pp. 1-15.

Non-Final Office Action, U.S. Appl. No. 15/201,442, May 4, 2018, 11 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,916, Apr. 7, 2021, 14 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,916, Apr. 6, 2022, 18 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,916, Sep. 16, 2019, 9 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,932, Dec. 27, 2019, 7 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,932, May 3, 2019, 11 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,937, Mar. 22, 2021, 14 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,937, Dec. 22, 2022, 21 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,937, Sep. 3, 2019, 21 pages.

Non-Final Office Action, U.S. Appl. No. 15/858,947, May 6, 2019, 7 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,268, Feb. 2, 2023, 14 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,268, Mar. 2, 2022, 15 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,268, Oct. 1, 2020, 10 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,268, May 28, 2019, 8 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,271, Mar. 9, 2021, 21 pages.

(56) **References Cited**

OTHER PUBLICATIONS

Non-Final Office Action, U.S. Appl. No. 15/859,271, Mar. 2, 2022, 20 pages.

Non-Final Office Action, U.S. Appl. No. 15/859,271, Sep. 12, 2019, 9 pages.

Eiron et al., "Matrix Multiplication: A Case Study of Algorithm Engineering", 2nd Workshop on Algorithm Engineering (WAE98), 1998, pp. 98-109.

Joerg-Henrik Heine, "RDocumentation—Pair—Rasch Item Parameter", RDocumentation, Available Online at <https://www.rdocumentation.org/packages/pairwise/versions/0.3.1/topics/pair>, Version 0.3.1, 2015, 1 page.

Lahr Dave, "Timing Matrix Multiplication in SciDB and Setting the Number of Worker Instances in SciDB and Running Matrix Multiplication Piecemeal", Available Online at <http://dllahr.blogspot.com/2012/11/timing-matrix-multiplication-in-scidb.html>, Nov. 13, 2012, 8 pages.

Non-Final Office Action, U.S. Appl. No. 18/186,710, Jul. 17, 2024, 11 pages.

Non-Final Office Action, U.S. Appl. No. 18/386,407, Jun. 26, 2024, 7 pages.
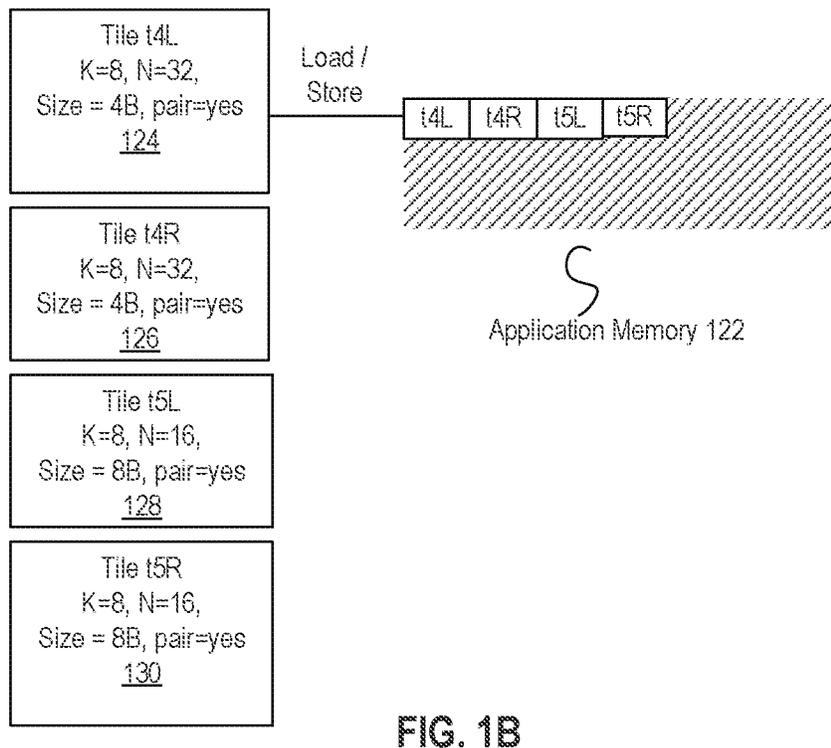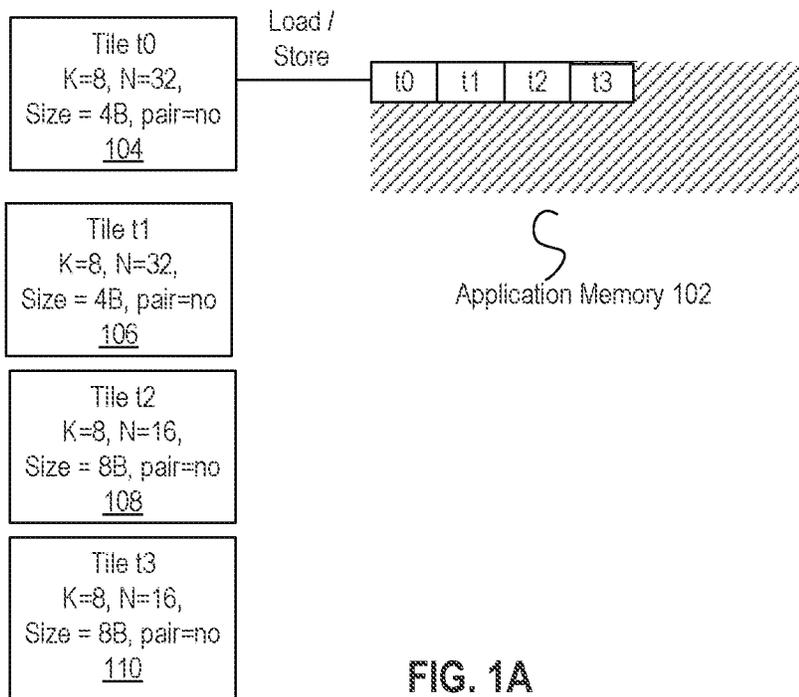
Notice of Allowance and Fees Due (PTOL-85) Mailed on Jan. 27, 2021 for U.S. Appl. No. 15/858,947, 9 pages.

Office Action, CN App. No. 202111269207.5, Nov. 29, 2024, 07 pages of Original document only.

Okuyama et al., "Design of a Coarse-Grained Processing Element for Matrix Multiplication on FPGA", 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, Sep. 23-25, 2014, pp. 237-241.

Reviriego et al., "An Efficient Technique to Protect Serial Shift Registers Against Soft Errors", IEEE Transactions On Circuits and Systems—II: Express Briefs, vol. 60, No. 8, Aug. 2013, 5 pages.
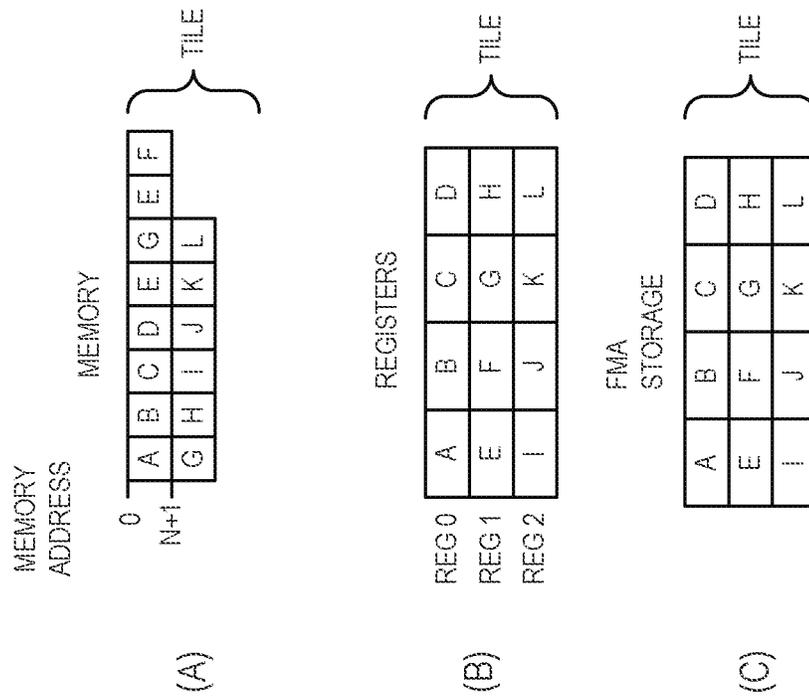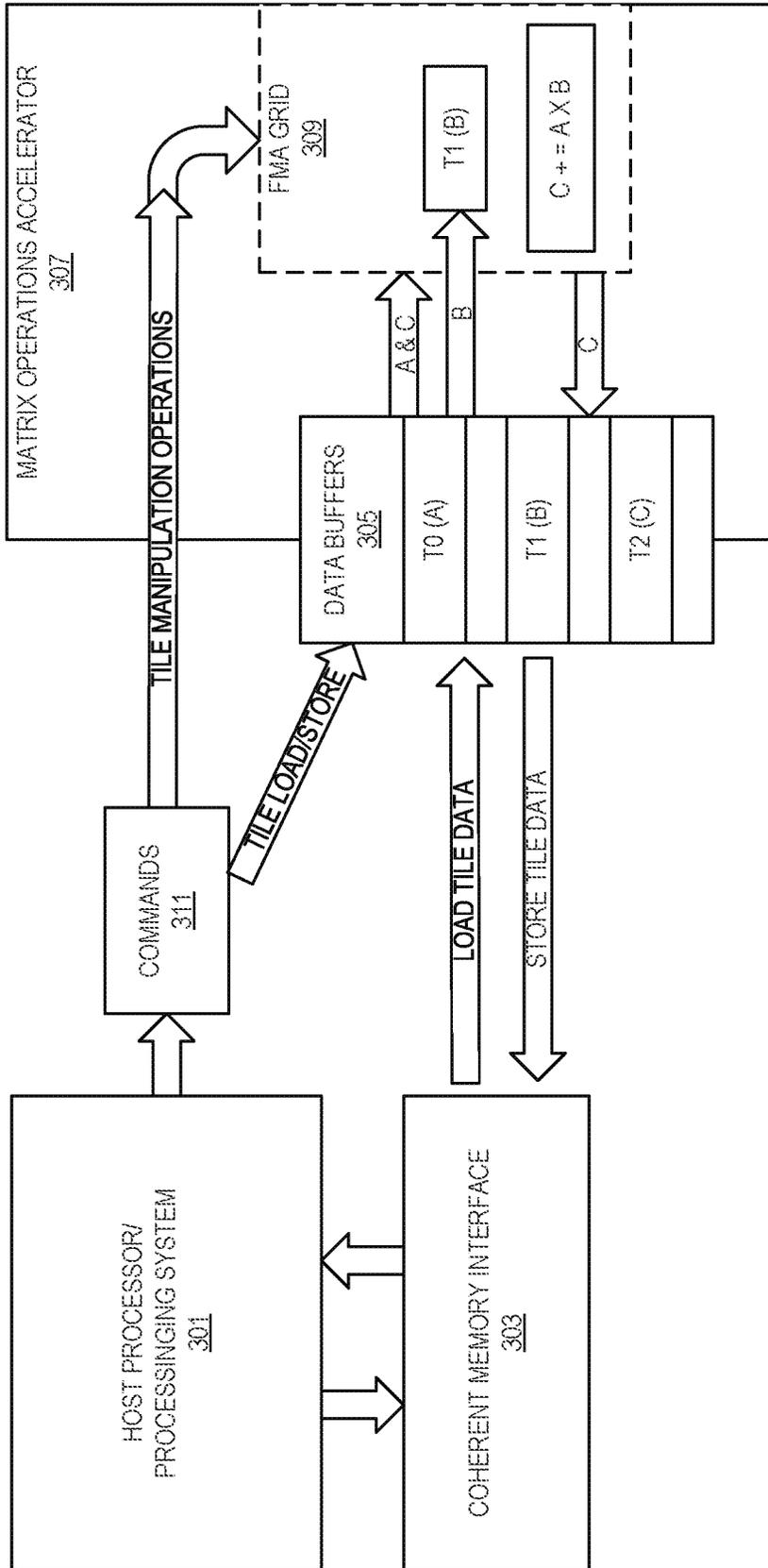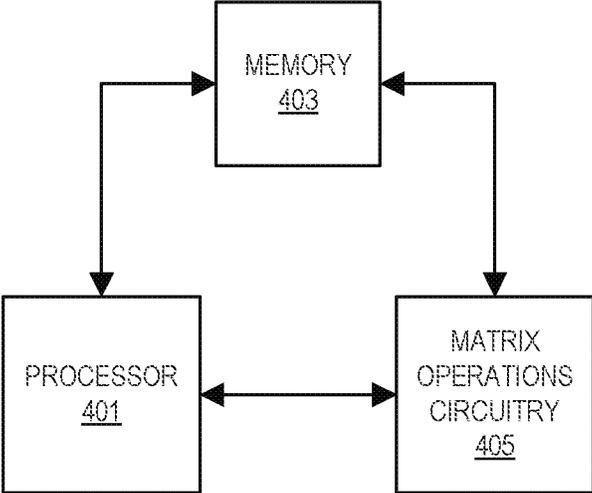
* cited by examiner

Tile t0
K=8, N=32,
Size = 4B, pair=no
104

Load /
Store

| t0 | t1 | t2 | t3 |

Application Memory 102

Tile t1
K=8, N=32,
Size = 4B, pair=no
106

Tile t2
K=8, N=16,
Size = 8B, pair=no
108

Tile t3
K=8, N=16,
Size = 8B, pair=no
110

FIG. 1A

Tile t4L
K=8, N=32,
Size = 4B, pair=yes
124

Load /
Store

| t4L | t4R | t5L | t5R |

Application Memory 122

Tile t4R
K=8, N=32,
Size = 4B, pair=yes
126

Tile t5L
K=8, N=16,
Size = 8B, pair=yes
128

Tile t5R
K=8, N=16,
Size = 8B, pair=yes
130

FIG. 1B

FIG. 2

(A) MEMORY

MEMORY ADDRESS

| 0 | A | B | C | D | E | G | E | F |
| N+1 | G | H | I | J | K | L |

TILE

(B) REGISTERS

REG 0	A	B	C	D
REG 1	E	F	G	H
REG 2	I	J	K	L

TILE

(C) FMA STORAGE

A	B	C	D
E	F	G	H
I	J	K	L

TILE

FIG. 3

FIG. 4

FIG. 5

FIG. 6

FIG. 7

SIGNED
SOURCE 1
ELEMENTS
801

3	2
3	2

SIGNED
SOURCE 2
ELEMENTS
803

3	2
0	1

MULTIPLIER
805

MULTIPLIER
807

INIT/
PREVIOUS
RESULT
809

1

ADDER/
SATURATION
813

ITERATION
RESULT
815

3

FIG. 8

FIG. 9

FIG. 10

ACCUMULATOR 2X INPUT SIZES 1101

SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	WORD/HPFP	16
WORD	16	INT32/SPFP	32
SPFP/INT32	32	INT64/DPFP	64

ACCUMULATOR 4X INPUT SIZES 1103

SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	INT32/SPFP	32
WORD	16	INT64/DPFP	64

ACCUMULATOR 8X INPUT SIZES 1105

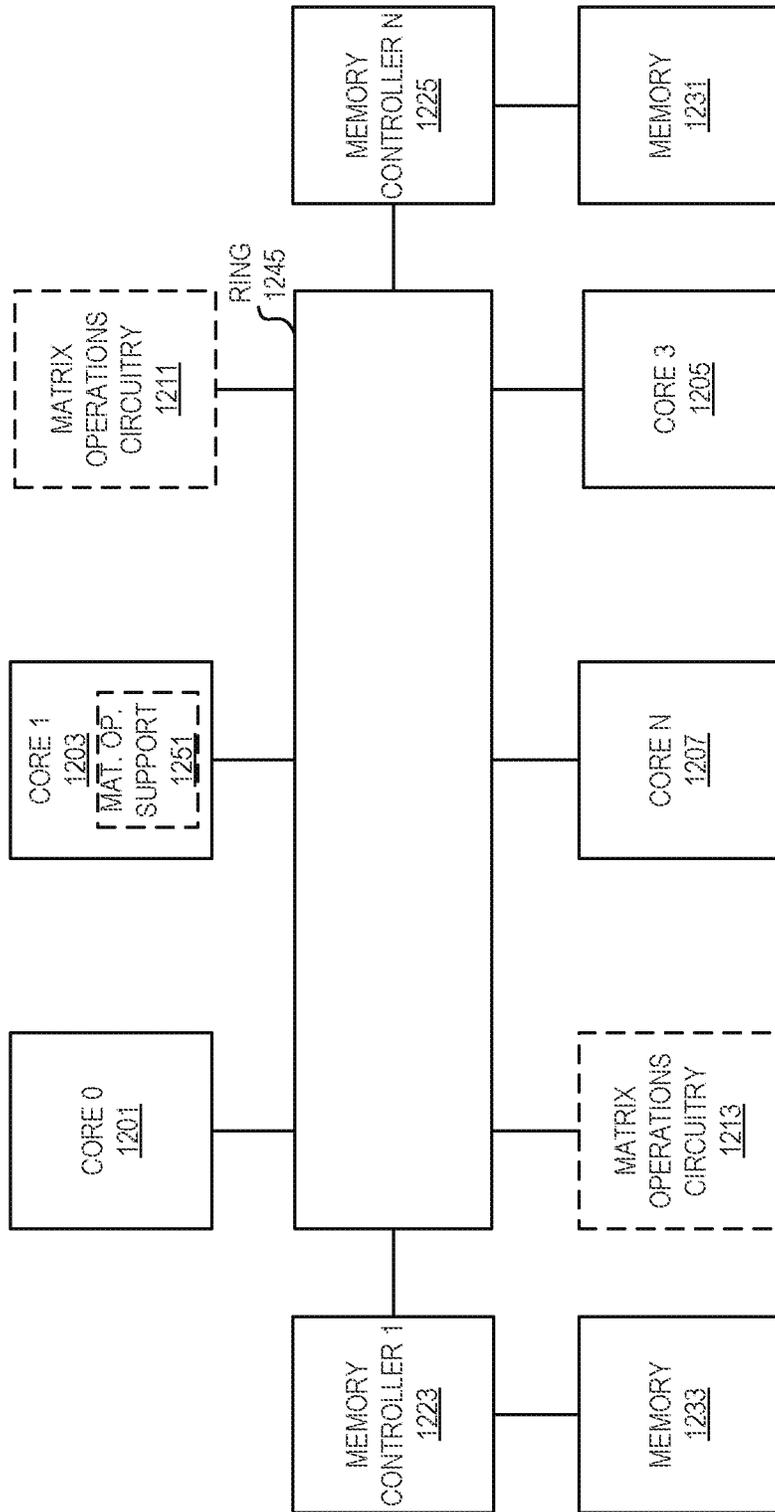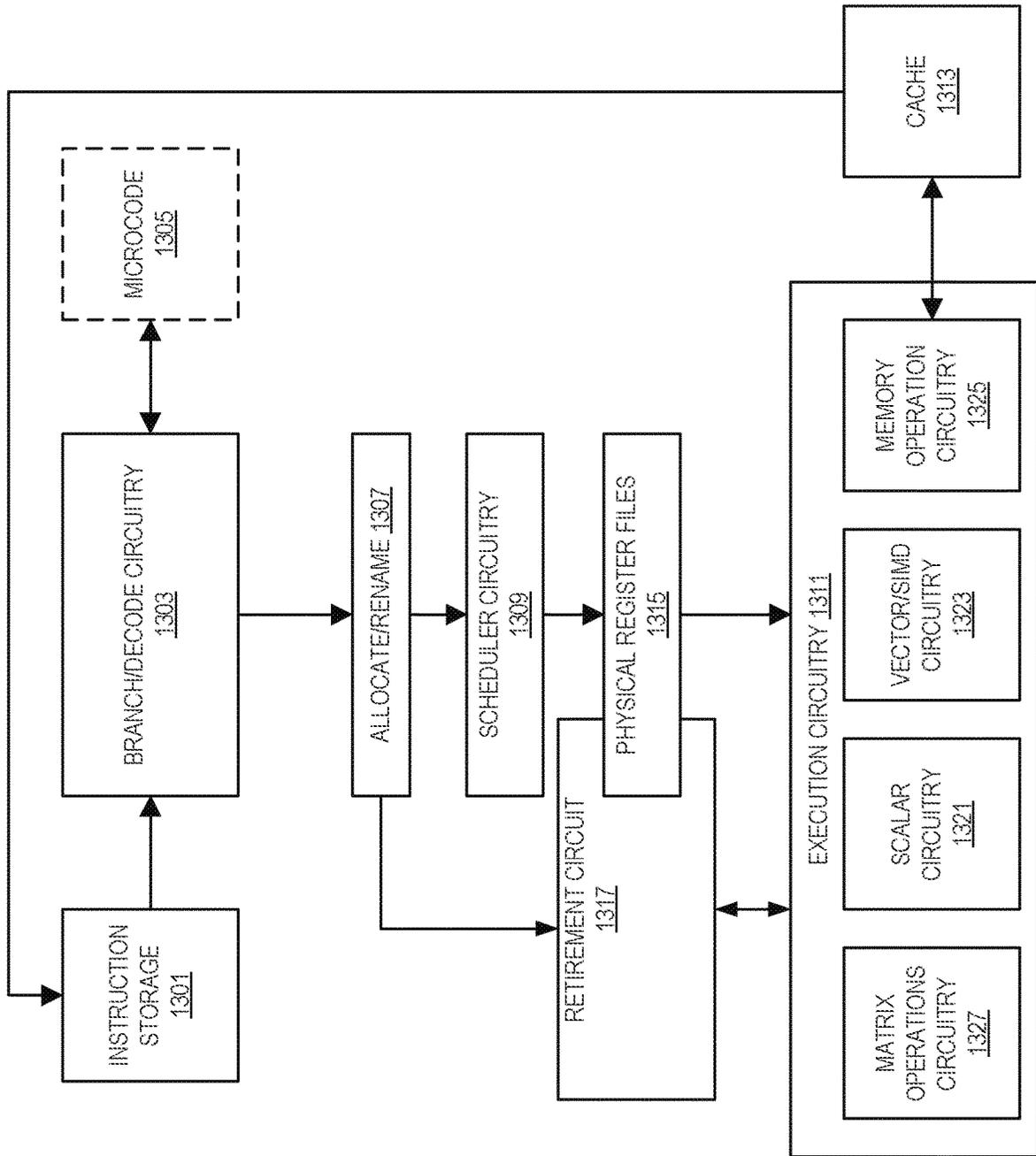| SOURCES	BITS	ACCUMULATOR	BITS
BYTE	8	INT64/DPFP	64

FIG. 11

FIG. 12

FIG. 13
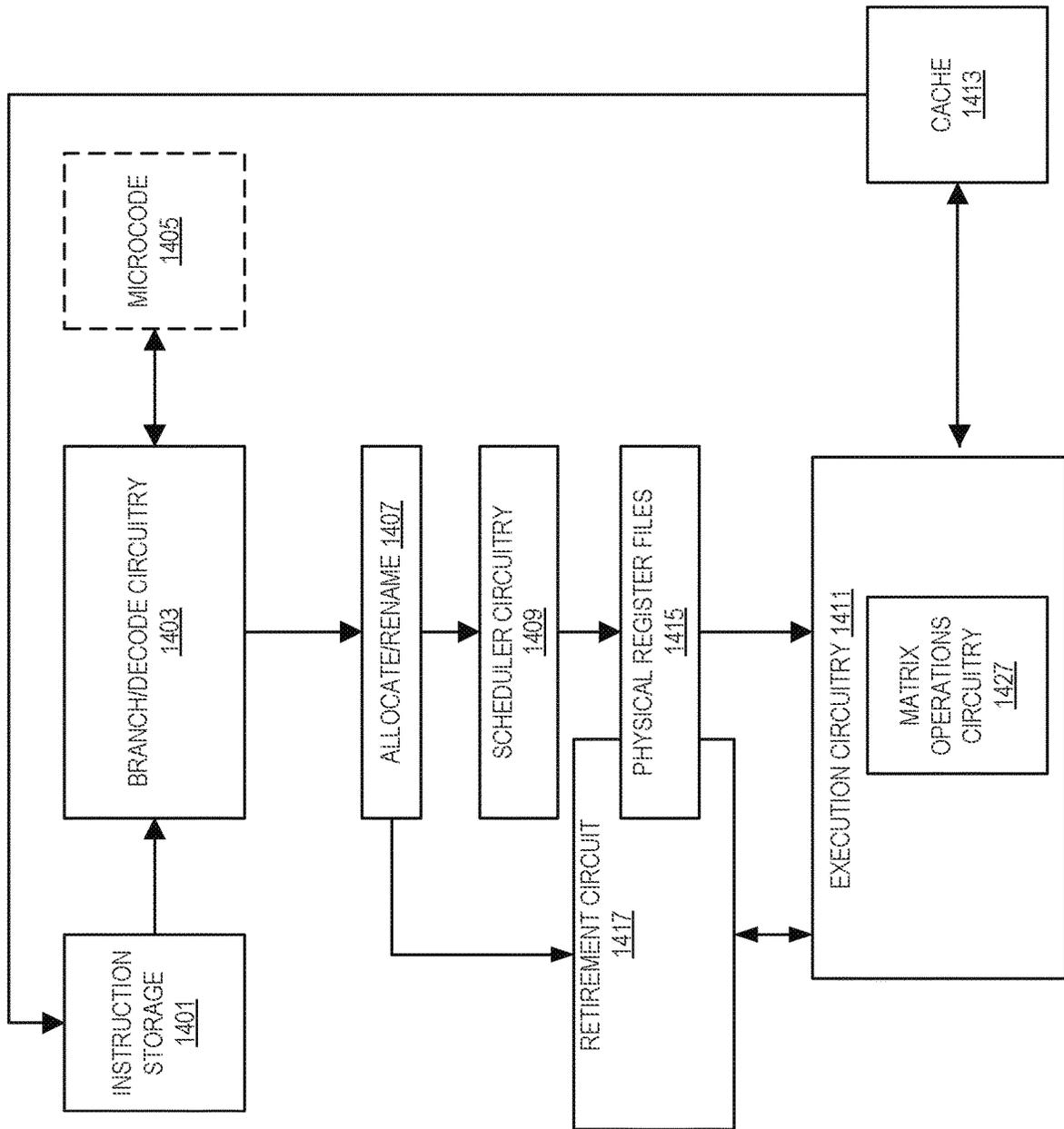
FIG. 14

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

ADDR	VALUE
0	A_{11}
1	A_{12}
2	A_{13}
3	A_{21}
4	A_{22}
5	A_{23}

ROW MAJOR

ADDR	VALUE
0	A_{11}
1	A_{21}
2	A_{12}
3	A_{22}
4	A_{13}
5	A_{23}

COLUMN MAJOR

FIG. 15
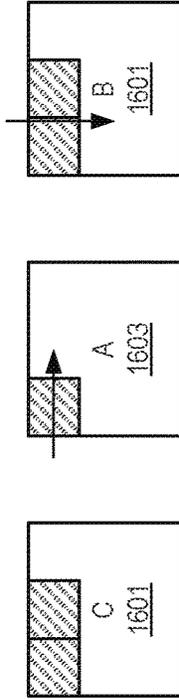
```
TILECONFIG [RAX]
// ASSUME SOME OUTER LOOPS DRIVING THE CACHE TILING (NOT SHOWN)
{
TILELOAD TMM0, RSI+RDI // SRCDST, RSI POINTS TO C, RDI HAS
TILELOAD TMM1, RSI+RDI+N // SECOND TILE OF C, UNROLLING IN SIMD DIMENSION N
MOV KK, 0
LOOP:
TILELOAD TMM2, R8+R9 // SRC2 IS STRIDED LOAD OF A, REUSED FOR 2 TMMA INSTR.
TILELOAD TMM3, R10+R11 // SRC1 IS STRIDED LOAD OF B
TMMAPS TMM0, TMM2, TMM3 // UPDATE LEFT TILE OF C
TILELOAD TMM3, R10+R11+N // SRC1 LOADED WITH B FROM NEXT RIGHTMOST TILE
TMMAPS TMM1, TMM2, TMM3 // UPDATE RIGHT TILE OF C
ADD R8, K // UPDATE POINTERS BY CONSTANTS KNOWN OUTSIDE OF LOOP
ADD R10, K*R11
ADD KK, K
CMP KK, LIMIT
JNE LOOP
TILESTORE RSI+RDI, TMM0 // UPDATE THE C MATRIX IN MEMORY
TILESTORE RSI+RDI+M, TMM1
} // END OF OUTER LOOP
TILERELEASE // RETURN TILES TO INIT STATE
```

FIG. 16

CONFIGURE USAGE OF MATRICES (TILES) 1701

LOAD AT LEAST ONE MATRIX (TILE) FROM MEMORY 1703

PERFORM AT LEAST ONE MATRIX (TILE) OPERATION 1705

STORE AT LEAST ONE MATRIX (TILE) TO MEMORY 1707

CONTEXT SWITCH 1709
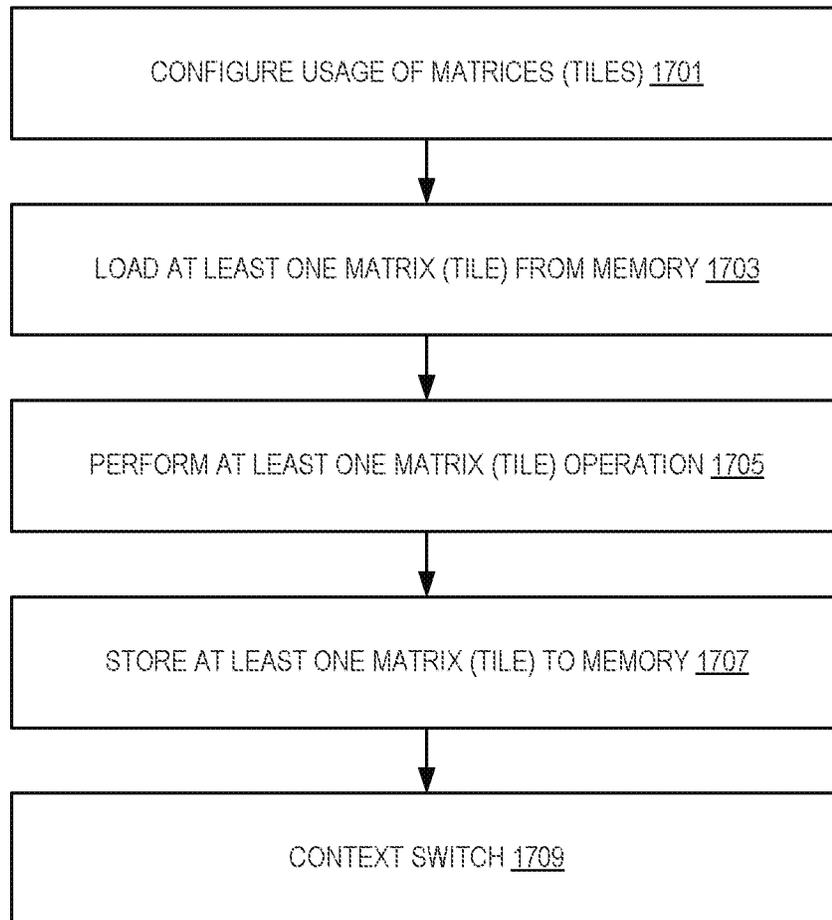
FIG. 17
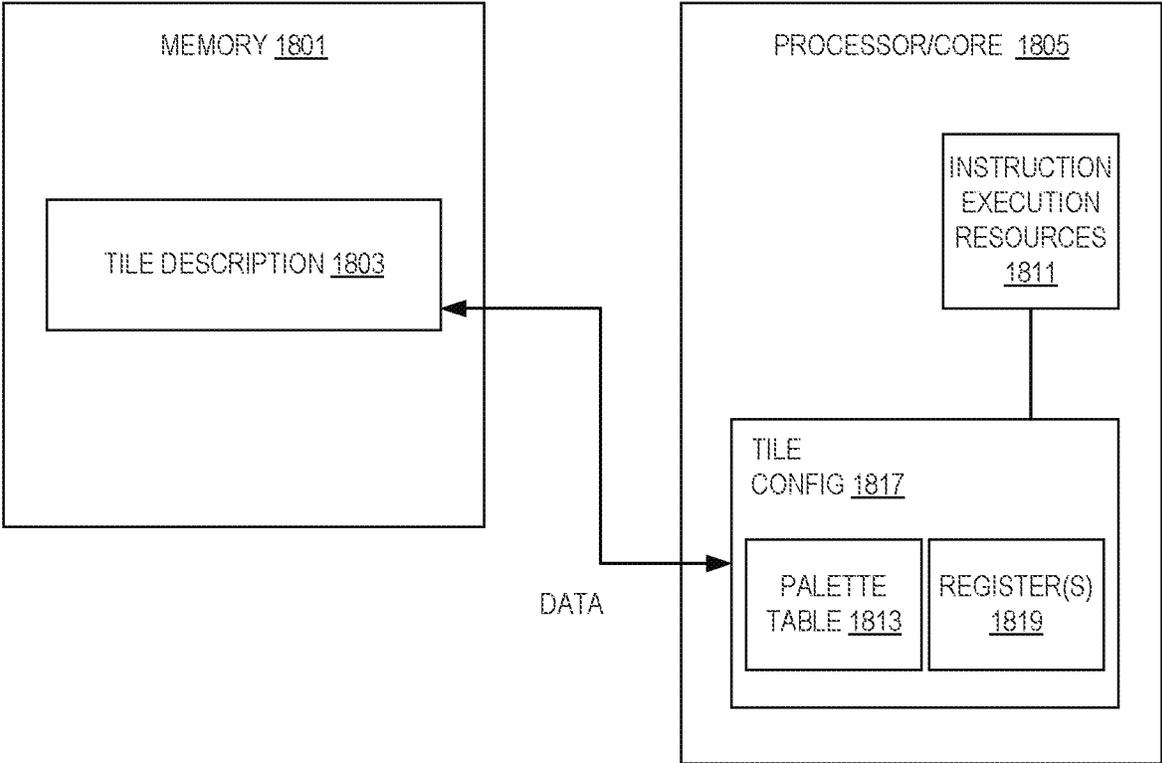
FIG. 18

PALETTE ID 1901	STARTM 1903
STARTP 1905	PAIR INDICATORS 1907
0	0
0	0

. . .

0	0
TMM0 ROWS 1913	TMM0 COLUMNS 1915
TMM1 ROWS	TMM1 COLUMNS
▪ ▪ ▪	
TMM15 ROWS	TMM15 COLUMNS
0	

FIG. 19

REGISTERS 1819

TMM0 CONFIG 2001	STARTP 2011	
. . .		STATUS 2015
TMMN CONFIG 2003	STARTROW 2013	

FIG. 20(A)

REGISTERS 1819

TMM0 ROW CONFIG 2021	TMM0 COL. CONFIG 2023	STARTP 2011	
. . .	. . .		STATUS 2015
TMMN ROW CONFIG	TMMN COL. CONFIG	STARTROW 2013	

FIG. 20(B)

REGISTERS 1819

TMM CONFIGS 2031	STARTP 2011	STARTROW 2013	STATUS 2015

FIG. 20(C)

REGISTERS 1819

TMM CONFIGS 2031		
STARTP 2011	STARTROW 2013	STATUS 2015

FIG. 20(D)

FETCH AN INSTRUCTION HAVING AN OPCODE AND A FIELD FOR A MEMORY LOCATION 2101

↓

DECODE THE INSTRUCTION 2103

↓

SCHEDULE THE DECODED INSTRUCTION AND RESOLVE AN ADDRESS ASSOCIATED WITH THE MEMORY LOCATION 2105

↓

EXECUTE THE DECODED INSTRUCTION TO:
RETRIEVE CONFIGURED MATRIX (TILE) DESCRIPTION DATA AND STORE THE RETRIVED DESCRIPTION DATA AT THE MEMORY LOCATION 2107

↓

COMMIT 2109

FIG. 21

YES $\Big\langle$ DETERMINE TILES USAGE IS CONFIGURED 2201 $\Big\rangle$ NO → WRITE ALL ZEROS AT MEMORY ADDRESS 2203

CREATE AND WRITE MATRIX (TILE) DESCRIPTION TO THE MEMORY ADDRESS 2205

1) THE PALETTE ID INTO A FIRST BYTE ASSOCIATED WITH THE MEMORY ADDRESS,
2) THE STARTROW VALUE INTO A SECOND BYTE ASSOCIATED WITHTHE MEMORY ADDRESS
3) THE STARTP VALUE INTO A THIRD BYTE ASSOCIATED WITHTHE MEMORY ADDRESS
4) THE PAIR INDICATORS INTO A FOURTH BYTE ASSOCIATED WITHTHE MEMORY ADDRESS
2207

ADD FOR A MATRIX (TILE) NUMBER (BEGINNING AT ZERO) BASED ON THE SPECIFIC GROUPING AND SUBJECT TO CONDITIONS:
   1) A NUMBER OF ROWS
   2) A NUMBER OF COLUMNS
2209

END 2215

NO $\Big\langle$ DETERMINE IF ALL CONFIGURATION INFORMATION ABOUT MATRICES (TILES) HAS BEEN ADDED 2211 $\Big\rangle$ YES
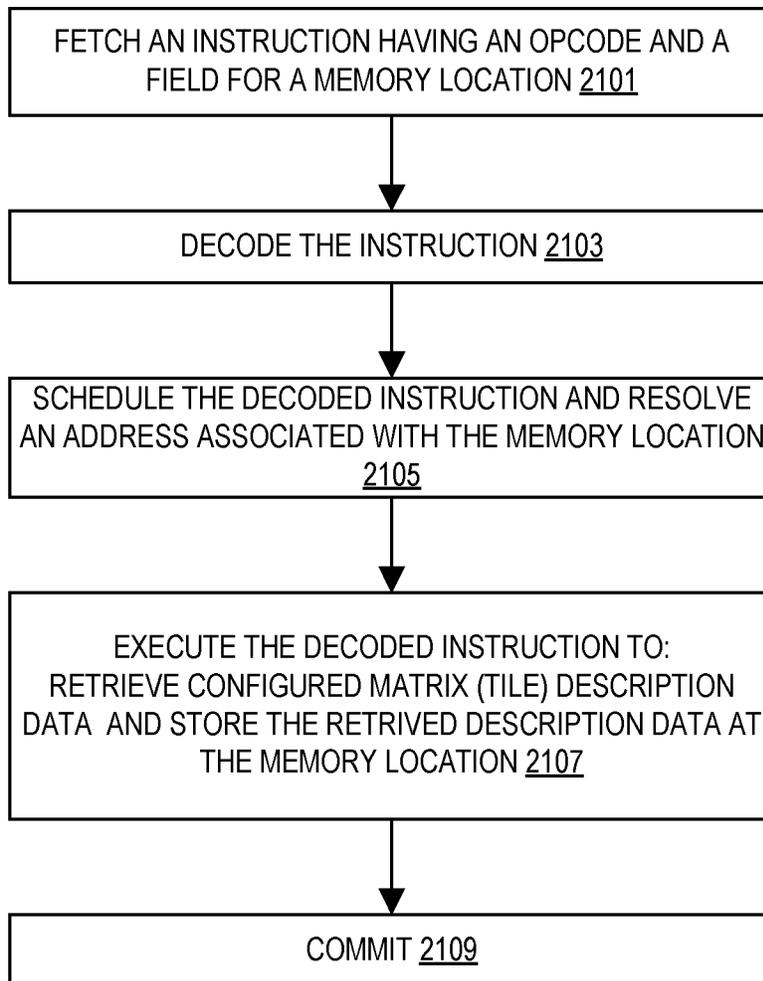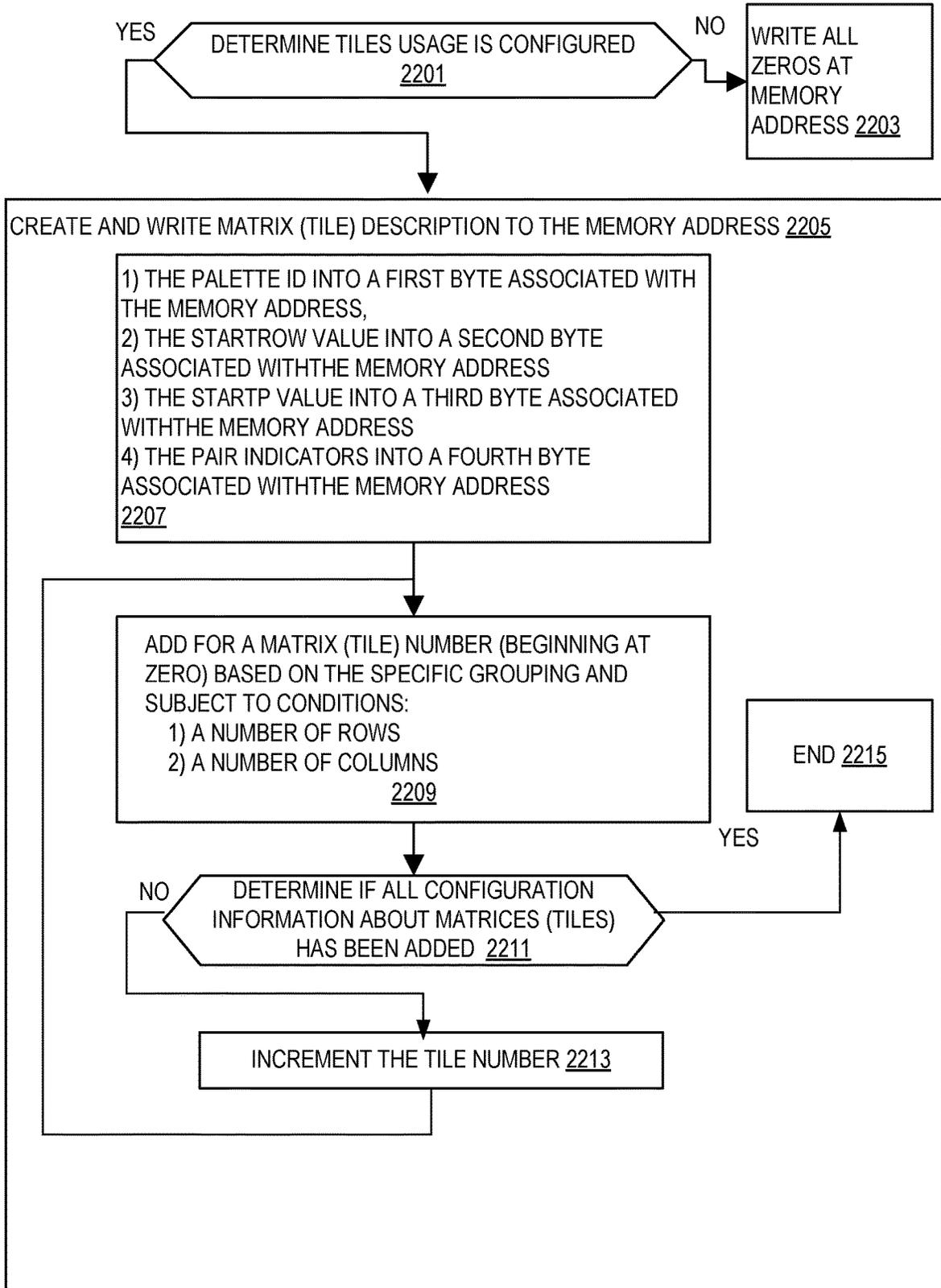
INCREMENT THE TILE NUMBER 2213

FIG. 22

```
// FORMAT OF MEMORY PAYLOAD. EACH FIELD IS A BYTE
// 0: PALETTE_ID
// 1: STARTROW
// 2: STARTP
// 3: PAIR INDICATORS
// 4-15: RESERVED (MUST BE ZERO)
// 16-17: TILE0ROWSTILE0COLS
// 18-19: TILE1ROWSTILE1COLS
// 20-21: TILE2ROWSTILE2COLS
// ...
// 46-47: TILE15ROWS TILE15COLS
// 48-63: 16B RESERVED (MUST BE ZERO)
STTILECFG MEM
IF TILES_CONFIGURED == 0:
//CREATE LOCAL BUFFER OF ZEROS AND COMMIT TO MEMORY
BUF[0..63] := 0
WRITE_MEMORY(MEM, 64, BUF)
ELSE:
MEM.BYTE[0] := TILECONFIG.PALETTE_ID
MEM.BYTE[1] := TILECONFIG.STARTROW
MEM.BYTE[2] := TILECONFIG.STARTP
MEM.BYTE[3] := TILECONFIG.PAIR_INDICATORS
MEM.BYTE[4..15] := 0
P := 16
FOR N IN 0 ... PALETTE_TABLE[TILECONFIG.PALETTE_ID].MAX_NAMES-1:
MEM.BYTE[P++] := TILECONFIG.T[N].ROWS
MEM.BYTE[P++] := TILECONFIG.T[N].COLS
WHILE P < 64:
 MEM.BYTE[P++] := 0
```
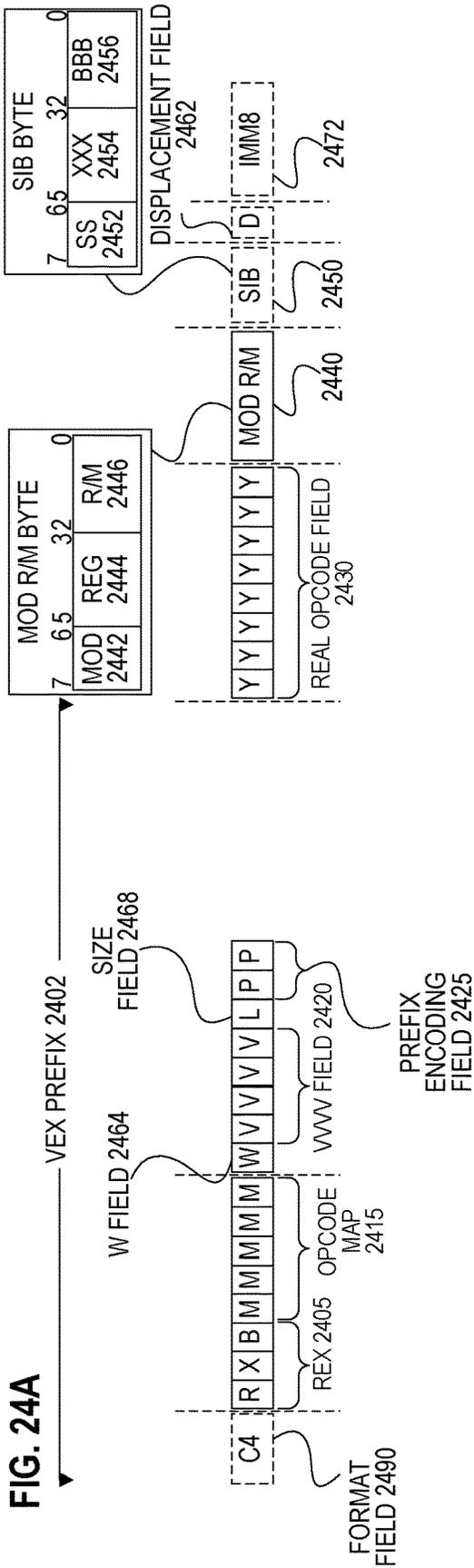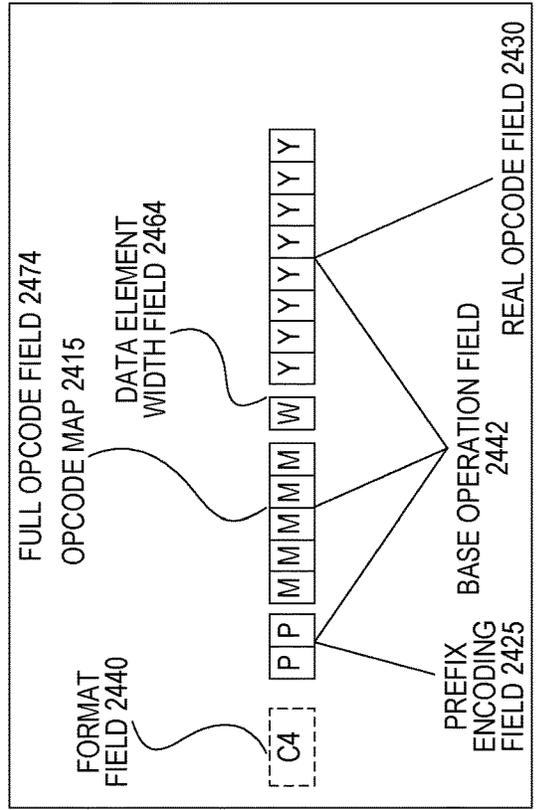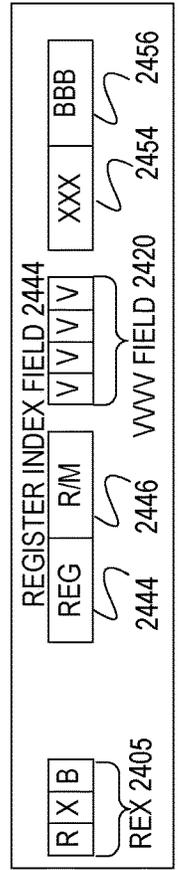
FIG. 23

FIG. 24A

FIG. 24B

FIG. 24C

FIG. 25

REGISTER ARCHITECTURE 2500

General Purpose Registers 2525
16 X 64 BITS

Vector Registers 2510
512 BITS

zmm₀   ymm₀   xmm₀

zmm₃₁   ymm₁₅   xmm₁₅

128 BITS

256 BITS

SCALAR FP STACK REGISTER FILE
2545
(X87FP)
80 BITS

ALIASED

0

7

64 BITS
MMX PACKED INT FLAT
REGISTER FILE 2550

TILES 2510

FIG. 26A

PIPELINE 2600

| FETCH 2602 | LENGTH DECODING 2604 | DECODE 2606 | ALLOC. 2608 | RENAMING 2610 | SCHEDULE 2612 | REGISTER READ/ MEMORY READ 2614 | EXECUTE STAGE 2616 | WRITE BACK/ MEMORY WRITE 2618 | EXCEPTION HANDLING 2622 | COMMIT 2624 |

FIG. 26B

CORE 2690

FRONT END UNIT 2630

BRANCH PREDICTION UNIT 2632

INSTRUCTION CACHE UNIT 2634

INSTRUCTION TLB UNIT 2636

INSTRUCTION FETCH 2638

DECODE UNIT 2640

EXECUTION ENGINE UNIT 2650

RENAME / ALLOCATOR UNIT 2652

RETIREMENT UNIT 2654

SCHEDULER UNIT(S) 2656

PHYSICAL REGISTER FILES UNIT(S) 2658

EXECUTION CLUSTER(S) 2660

EXECUTION UNIT(S) 2662

MEMORY ACCESS UNIT(S) 2664

MEMORY UNIT 2670

DATA TLB UNIT 2672

DATA CACHE UNIT 2674

L2 CACHE UNIT 2676

FIG. 27B

16-WIDE VECTOR ALU 2728

SWIZZLE 2720

VECTOR REGISTERS 2714

NUMERIC CONVERT 2722B

L1 DATA CACHE 2706A

REPLICATE 2724

NUMERIC CONVERT 2722A

FIG. 27A

INSTRUCTION DECODE 2700

VECTOR UNIT 2710

SCALAR UNIT 2708

VECTOR REGISTERS 2714

SCALAR REGISTERS 2712

L1 CACHE 2706

LOCAL SUBSET OF THE L2 CACHE 2704

RING NETWORK 2702

FIG. 28

2900

2915

2910

PROCESSOR

2995

2945

2940

CONTROLLER
HUB 2920

MEMORY

CO-
PROCESSOR

GMCH 2990

2960

I/O

IOH 2950

FIG. 29

FIG. 30

FIG. 31

FIG. 32

SYSTEM ON A CHIP
3200

SYSTEM AGENT UNIT 2810

APPLICATION PROCESSOR 3210

CORE 2802A

CACHE UNIT(S) 2804A

CORE 2802N

CACHE UNIT(S) 2804N

SHARED CACHE UNIT(S) 2806

INTERCONNECT UNIT(S) 3202

BUS CONTROLLER UNIT(S) 2816

COPROCESSOR(S) 3220

INTEGRATED MEMORY CONTROLLER UNIT(S) 2814

SRAM UNIT 3230

DMA UNIT 3232

DISPLAY UNIT 3240

FIG. 33

PROCESSOR WITH AT LEAST ONE FIRST INSTRUCTION SET CORE 3316

PROCESSOR WITHOUT AN FIRST INSTRUCTION SET CORE 3314

HARDWARE

SOFTWARE

FIRST BINARY CODE 3306

FIRST COMPILER 3304

INSTRUCTION CONVERTER 3312

HIGH LEVEL LANGUAGE 3302

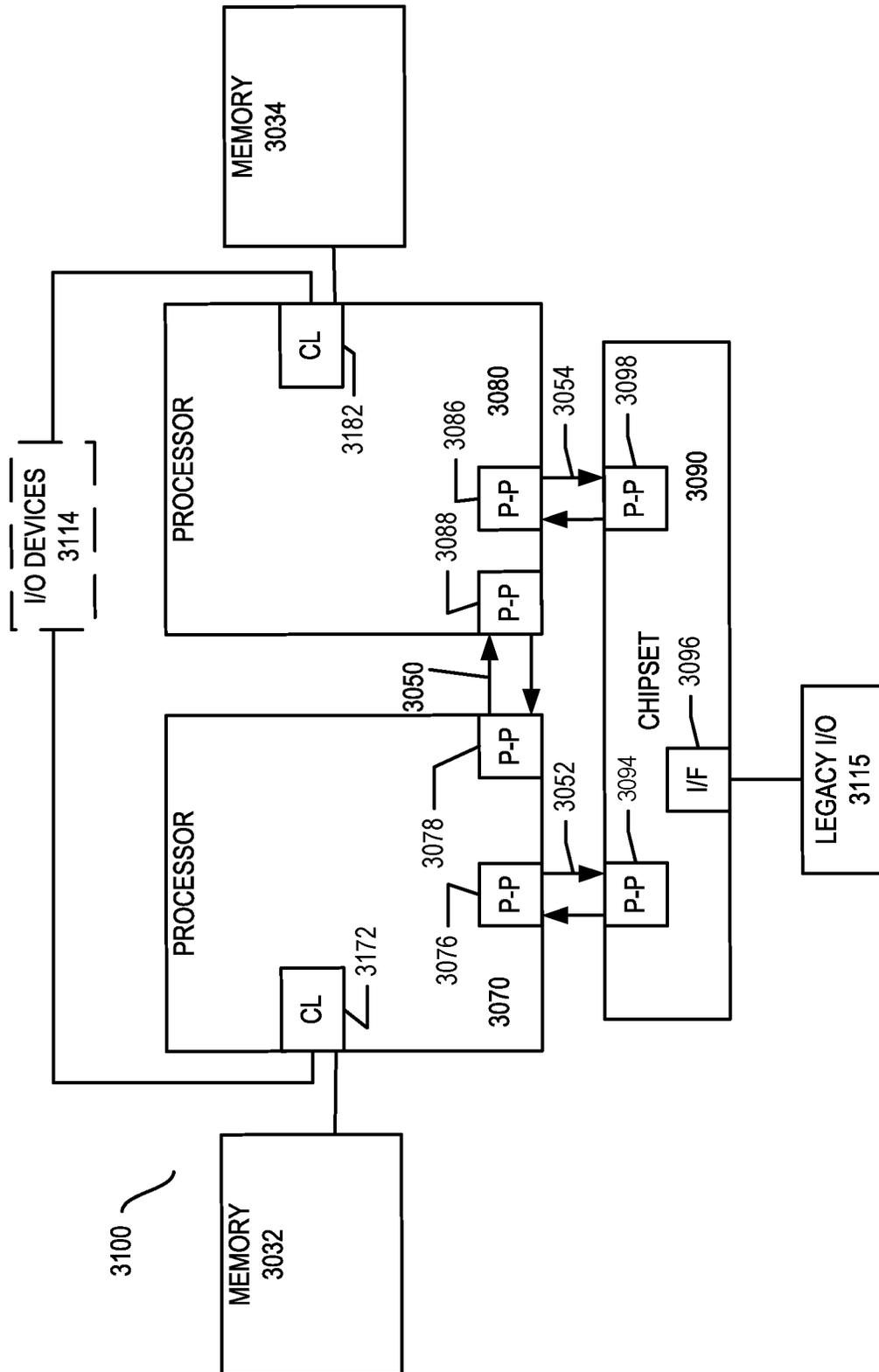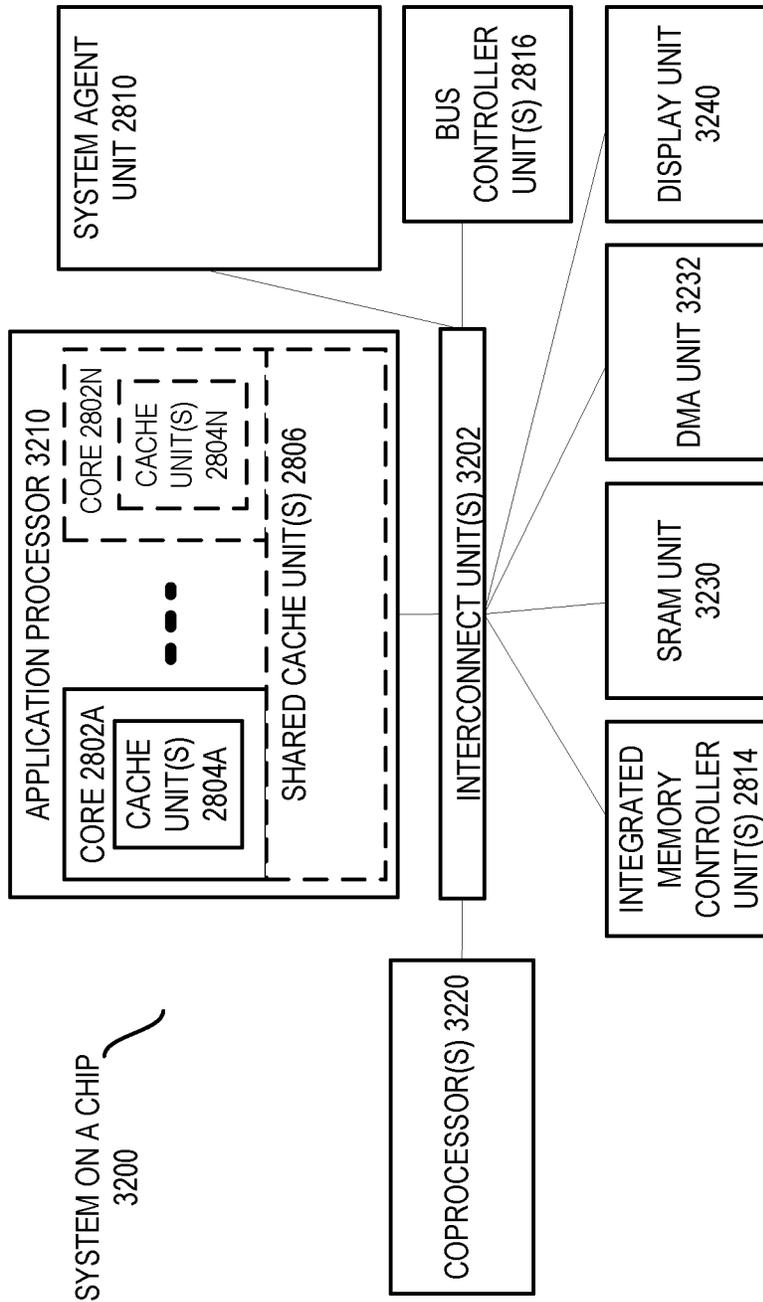ALTERNATIVE INSTRUCTION SET BINARY CODE 3310
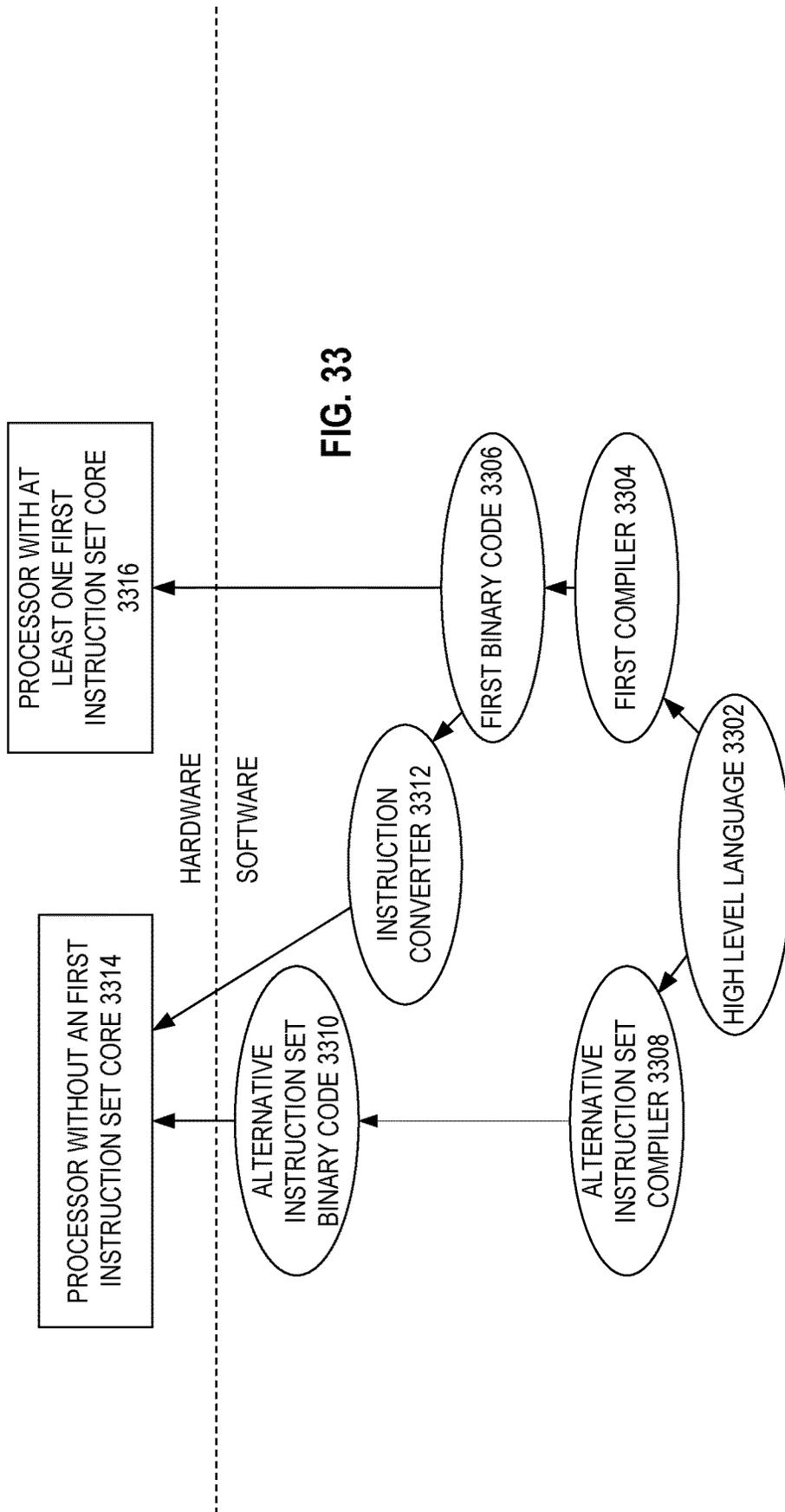
ALTERNATIVE INSTRUCTION SET COMPILER 3308

SYSTEMS, METHODS, AND APPARATUSES FOR MATRIX OPERATIONS

FIELD OF INVENTION

The field of invention relates generally to computer processor architecture, and, more specifically, to matrix manipulation.

BACKGROUND

Matrices are increasingly important in many computing tasks such as machine learning and other bulk data processing.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

FIGS. 1A and 1B illustrate an embodiment of configured tiles;

FIG. 2 illustrates several examples of matrix storage;

FIG. 3 illustrates an embodiment of a system utilizing a matrix (tile) operations accelerator;

FIGS. 4 and 5 show different embodiments of how memory is shared using a matrix operations accelerator;

FIG. 6 illustrates an embodiment of matrix multiply accumulate operation using tiles ("TMMA");

FIG. 7 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction;

FIG. 8 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction;

FIG. 9 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction;

FIG. 10 illustrates an embodiment of a subset of the execution of an iteration of chained fused multiply accumulate instruction;

FIG. 11 illustrates power-of-two sized SIMD implementations wherein the accumulators use input sizes that are larger than the inputs to the multipliers according to an embodiment;

FIG. 12 illustrates an embodiment of a system utilizing matrix operations circuitry;

FIG. 13 illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles;

FIG. 14 illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles;

FIG. 15 illustrates an example of a matrix expressed in row major format and column major format;

FIG. 16 illustrates an example of usage of matrices (tiles);

FIG. 17 illustrates an embodiment a method of usage of matrices (tiles);

FIG. 18 illustrates an exemplary execution of a STTILECFG instruction;

FIG. 19 illustrates an embodiment of a description of the matrices (tiles) to be supported;

FIGS. 20(A)-(D) illustrate examples of register(s);

FIG. 21 illustrates an embodiment of method performed by a processor to process a STTILECFG instruction;

FIG. 22 illustrates a more detailed description of an execution of a STTILECFG instruction using memory addressing;

FIG. 23 illustrates exemplary pseudocode for an execution of a STTILECFG instruction;

FIGS. 24(A)-(C) illustrate an exemplary instruction format;

FIG. 25 is a block diagram of a register architecture according to one embodiment of the invention;

FIGS. 26A-B illustrate the in-order pipeline and in-order core;

FIGS. 27A-B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip;

FIG. 28 is a block diagram of a processor 2800 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention;

FIGS. 29-32 are block diagrams of exemplary computer architectures; and

FIG. 33 is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention.

DETAILED DESCRIPTION

In the following description, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

References in the specification to "one embodiment," "an embodiment," "an example embodiment," etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

In many mainstream processors, handling matrices is a difficult and/or instruction intensive task. For example, rows of a matrix could be put into a plurality of packed data (e.g., SIMD or vector) registers and then operated on individually. For example, an add two 8×2 matrices may require a load or gather into four packed data registers depending upon data sizes. Then a first add of packed data registers corresponding to a first row from each matrix is performed and a second add of packed data registers corresponding to a second row from each matrix is performed. Then the resulting packed data registers are scattered back to memory. While for small matrices this scenario may be acceptable, it is often not acceptable with larger matrices.

I. High-Level Discussion

Described herein are mechanisms to support matrix operations in computer hardware such as central processing units (CPUs), graphic processing units (GPUs), and accelerators. The matrix operations utilize 2-dimensional (2-D) data structures representing one or more packed regions of memory such as registers. Throughout this description, these 2-D data structures are referred to as tiles. Note that a matrix may be smaller than a tile (use less than all of a tile), or

utilize a plurality of tiles (the matrix is larger than the size of any one tile). Throughout the description, matrix (tile) language is used to indicate operations performed using tiles that impact a matrix; whether or not that matrix is larger than any one tile is not typically relevant.

Each tile may be acted upon by different operations such as those that are detailed herein and include, but are not limited to: matrix (tile) multiplication, tile add, tile subtract, tile diagonal, tile zero, tile transpose, tile dot product, tile broadcast, tile row broadcast, tile column broadcast, tile multiplication, tile multiplication and accumulation, tile move, etc. Additionally, support for operators such as the use of a scale and/or bias may be used with these operations or in support of non-numeric applications in the future, for instance, OpenCL "local memory," data compression/decompression, etc.

Portions of storage (such as memory (non-volatile and volatile), registers, cache, etc.) are arranged into tiles of different horizontal and vertical dimensions. For example, a tile may have horizontal dimension of 4 (e.g., four rows of a matrix) and a vertical dimension of 8 (e.g., 8 columns of the matrix). Typically, the horizontal dimension is related to element sizes (e.g., 2-, 4-, 8-, 16-, 32-, 64-, 128-bit, etc.). Multiple datatypes (single precision floating point, double precision floating point, integer, etc.) may be supported.

A. Exemplary Usage of Configured Tiles

FIG. 1A illustrates an embodiment of configured tiles. As shown, 4 kB of application memory 102 have stored thereon 4 1 kB tiles, tile 0 104, tile 1 106, tile 2 108, and tile 3 110. In this example, the 4 tiles do not consist of pairs, and each have elements arranged in rows and columns. Tile t0 104 and tile t1 106 have K rows and N columns of 4-byte elements (e.g., single precision data), where K equals 8 and N=32. Tile t2 108 and tile t3 110 have K rows and N/2 columns of 8-byte elements (e.g., double precision data). As the double precision operands are twice the width of single precision, this configuration is consistent with a palette, used to provide tile options, supplying at least 4 names with total storage of at least 4 kB. In operation, the tiles can be loaded from and stored to memory using load and store operations. Depending upon the instruction encoding scheme used, the amount of available application memory, as well as the size, number, and configuration of available tiles varies.

FIG. 1B illustrates an embodiment of configured tiles. As shown, 4 kB of application memory 122 have stored thereon 2 pairs of 1 kB-tiles, the first pair being tile t4L 124 and tile t4R 126, and the second pair being tile t5L 128 and tile t5R 130. As shown, the pairs of tiles are divided into a left tile and a right tile. In other embodiments, the pair of tiles are divided into an even tile and an odd tile. In this example, the 4 tiles each have elements arranged in rows and columns. Tile t4L 124 and tile t4R 126 have K rows and N columns of 4-byte elements (e.g., single precision data), where K equals 8 and N equals 32. Tile t5L 128 and tile t5R 130 have K rows and N/2 columns of 8-byte elements (e.g., double precision data). As the double precision operands are twice the width of single precision, this configuration is consistent with a palette, used to provide tile options, supplying at least 2 names with total storage of at least 4 kB. The four tiles of FIG. 1A use 4 names, each naming a 1 kB tile, whereas the 2 pairs of tiles in FIG. 1B can use 2 names to specify the paired tiles. In some embodiments, tile instructions accept a name of a paired tile as an operand. In operation, the tiles can be loaded from and stored to memory using load and store operations. Depending upon the instruction encoding

scheme used, the amount of available application memory, as well as the size, number, and configuration of available tiles varies.

In some embodiments, tile parameters are definable. For example, a "palette" is used to provide tile options. Exemplary options include, but are not limited to: the number of tile names, the number of bytes in a row of storage, the number of rows and columns in a tile, etc. For example, a maximum "height" (number of rows) of a tile may be defined as:

$$Tile\ Max\ Rows = Architected\ Storage/(The\ Number\ of\ Palette\ Names*The\ Number\ of\ Bytes\ per\ row)$$

As such, an application can be written such that a fixed usage of names will be able to take advantage of different storage sizes across implementations.

Configuration of tiles is done using a tile configuration ("TILECONFIG") instruction, where a particular tile usage is defined in a selected palette. This declaration includes the number of tile names to be used, the requested number of rows and columns per name (tile), and, in some embodiments, the requested datatype of each tile. In some embodiments, consistency checks are performed during the execution of a TILECONFIG instruction to determine that it matches the restrictions of the palette entry.

B. Exemplary Tile Storage Types

FIG. 2 illustrates several examples of matrix storage. In (A), a tile is stored in memory. As shown, each "row" consists of four packed data elements. To get to the next "row," a stride value is used. Note that rows may be consecutively stored in memory. Strided memory accesses allows for access of one row to then next when the tile storage does not map the underlying memory array row width.

Tile loads from memory and stores to memory are typically strided accesses from the application memory to packed rows of data. Exemplary TILELOAD and TILE-STORE instructions, or other instruction references to application memory as a TILE operand in load-op instructions, are, in some embodiments, restartable to handle (up to) 2*rows of page faults, unmasked floating point exceptions, and/or interrupts per instruction.

In (B), a matrix is stored in a tile comprised of a plurality of registers such as packed data registers (single instruction, multiple data (SIMD) or vector registers). In this example, the tile is overlaid on three physical registers. Typically, consecutive registers are used, however, this need not be the case.

In (C), a matrix is stored in a tile in non-register storage accessible to a fused multiple accumulate (FMA) circuit used in tile operations. This storage may be inside of a FMA, or adjacent to it. Additionally, in some embodiments, discussed below, the storage may be for a data element and not an entire row or tile.

The supported parameters for the TMMA architecture are reported via CPUID. In some embodiments, the list of information includes a maximum height and a maximum SIMD dimension. Configuring the TMMA architecture requires specifying the dimensions for each tile, the element size for each tile and the palette identifier. This configuration is done by executing the TILECONFIG instruction.

Successful execution of a TILECONFIG instruction enables subsequent TILE operators. A TILERELEASEALL instruction clears the tile configuration and disables the TILE operations (until the next TILECONFIG instructions executes). In some embodiments, XSAVE, XSTORE, etc.

are used in context switching using tiles. In some embodiments, 2 XCRO bits are used in XSAVE, one for TILE-CONFIF metadata and one bit corresponding to actual tile payload data.

TILECONFIG not only configures the tile usage, but also sets a state variable indicating that the program is in a region of code with tiles configured. An implementation may enumerate restrictions on other instructions that can be used with a tile region such as no usage of an existing register set, etc.

Exiting a tile region is typically done with the TILERE-LEASEALL instruction. It takes no parameters and swiftly invalidates all tiles (indicating that the data no longer needs any saving or restoring) and clears the internal state corresponding to being in a tile region.

In some embodiments, tile operations will zero any rows and any columns beyond the dimensions specified by the tile configuration. For example, tile operations will zero the data beyond the configured number of columns (factoring in the size of the elements) as each row is written. For example, with 64-byte rows and a tile configured with 10 rows and 12 columns, an operation writing FP32 elements would write each of the first 10 rows with 12*4 bytes with output/result data and zero the remaining 4*4 bytes in each row. Tile operations also fully zero any rows after the first 10 configured rows. When using 1 K tile with 64-byte rows, there would be 16 rows, so in this example, the last 6 rows would also be zeroed.

In some embodiments, a context restore (e.g., XRSTOR), when loading data, enforces that the data beyond the configured rows for a tile will be maintained as zero. If there is no valid configuration, all rows are zeroed. XRSTOR of tile data can load garbage in the columns beyond those configured. It should not be possible for XRSTOR to clear beyond the number of columns configured because there is not an element width associated with the tile configuration.

Context save (e.g., XSAVE) exposes the entire TILE storage area when writing it to memory. If XRSTOR loaded garbage data in to the rightmost part of a tile, that data will be saved by XSAVE. XSAVE will write zeros for rows beyond the number specified for each tile.

In some embodiments, tile instructions are restartable. The operations that access memory allow restart after page faults. The computational instructions that deal with floating point operations also allow for unmasked floating point exceptions, with the masking of the exceptions controlled by a control and/or status register.

To support restarting instructions after event that cause a matrix (tile) operation to break during execution, the instructions store information in the start registers detailed below.

II. Matrix (Tile) Operation Systems

A. Exemplary Hardware Support

FIG. **3** illustrates an embodiment of a system utilizing a matrix (tile) operations accelerator. In this illustration, a host processor/processing system **301** communicates commands **311** (e.g., matrix manipulation operations such as arithmetic or matrix manipulation operations, or load and store operations) to a matrix operations accelerator **307**. However, this is shown this way for discussion purposes only. As detailed later, this accelerator **307** may be a part of a processing core. Typically, commands **311** that are tile manipulation operator instructions will refer to tiles as register-register ("reg-reg") or register-memory ("reg-mem") format. Other commands such as TILESTORE, TILELOAD, TILECONFIG, etc., do not perform data operations on a tile. Commands may be

decoded instructions (e.g., micro-ops) or macro-instructions for the accelerator **307** to handle.

In this example, a coherent memory interface **303** is coupled to the host processor/processing system **301** and matrix operations accelerator **405** such that they can share memory. FIGS. **4** and **5** show different embodiments of how memory is shared using a matrix operations accelerator. As shown in FIG. **4**, the host processor **401** and matrix operations accelerator circuitry **405** share the same memory **403**. FIG. **5** illustrates an embodiment where the host processor **501** and matrix operations accelerator **505** do not share memory, but can access each other's memory. For example, processor **501** can access tile memory **507** and utilize its host memory **503** as normal. Similarly, the matrix operations accelerator **505** can access host memory **503**, but more typically uses its own memory **507**. Note these memories may be of different types.

In some embodiments, the matrix operations accelerator **307** includes a plurality of FMAs **309** coupled to data buffers **305** (in some implementations, one or more of these buffers **305** are stored in the FMAs of the grid as shown). The data buffers **305** buffer tiles loaded from memory and/or tiles to be stored to memory (e.g., using a tileload or tilestore instruction). Data buffers may be, for example, a plurality of registers. Typically, these FMAs are arranged as a grid of chained FMAs **309** which are able to read and write tiles. In this example, the matrix operations accelerator **307** is to perform a matrix multiply operation using tiles T0, T1, and T2. At least one of tiles is housed in the FMA grid **309**. In some embodiments, all tiles in an operation are stored in the FMA grid **309**. In other embodiments, only a subset are stored in the FMA grid **309**. As shown, T1 is housed and T0 and T2 are not. Note that A, B, and C refer to the matrices of these tiles which may or may not take up the entire space of the tile.

FIG. **6** illustrates an embodiment of matrix multiply accumulate operation using tiles ("TMMA").

The number of rows in the matrix (TILE A **601**) matches the number of serial (chained) FMAs comprising the computation's latency. An implementation is free to recirculate on a grid of smaller height, but the computation remains the same.

The source/destination vector comes from a tile of N rows (TILE C **605**) and the grid of FMAs **611** performs N vector-matrix operations resulting in a complete instruction performing a matrix multiplication of tiles. Tile B **603** is the other vector source and supplies "broadcast" terms to the FMAs in each stage.

In operation, in some embodiments, the elements of matrix B (stored in a tile B **603**) are spread across the rectangular grid of FMAs. Matrix B (stored in tile A **601**) has its elements of a row transposed to match up with the columnar dimension of the rectangular grid of FMAs. At each FMA in the grid, an element of A and B are multiplied and added to the incoming summand (from above) and the outgoing sum is passed to the next row of FMAs (or the final output).

The latency of a single step is proportional to K (row height of matrix B) and dependent TMMAs typically have enough source-destination rows (either in a single tile or across tile) to hide that latency. An implementation may also split the SIMD (packed data element) dimension M (row height of matrix A) across time steps, but this simply changes the constant that K is multiplied by. When a program specifies a smaller K than the maximum enumerated by the TMACC, an implementation is free to implement this with "masking" or "early outs."

7
8

The latency of an entire TMMA is proportional to N*K. The repeat rate is proportional to N. The number of MACs per TMMA instruction is N*K*M.

FIG. 7 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on signed sources wherein the accumulator is 2×the input data size.

A first signed source (source 1 701) and a second signed source (source 2 703) each have four packed data elements. Each of these packed data elements stores signed data such as floating point data. A third signed source (source 3 709) has two packed data elements, each of which stores signed data. The sizes of the first and second signed sources 701 and 703 are half that of the third signed source (initial value or previous result) 709. For example, the first and second signed sources 701 and 703 could have 32-bit packed data elements (e.g., single precision floating point) while the third signed source 709 could have 64-bit packed data elements (e.g., double precision floating point).

In this illustration, only the two most significant packed data element positions of the first and second signed sources 701 and 703 and the most significant packed data element position of the third signed source 709 are shown. Of course, the other packed data element positions would also be processed.

As illustrated, packed data elements are processed in pairs. For example, the data of the most significant packed data element positions of the first and second signed sources 701 and 703 are multiplied using a multiplier circuit 705, and the data from second most significant packed data element positions of the first and second signed sources 701 and 703 are multiplied using a multiplier circuit 707. In some embodiments, these multiplier circuits 705 and 707 are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source 709. The results of each of the multiplications are added using addition circuitry 711.

The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of the signed source 3 709 (using a different adder 713 or the same adder 711).

Finally, the result of the second addition is either stored into the signed destination 715 in a packed data element position that corresponds to the packed data element position used from the signed third source 709, or passed on to the next iteration, if there is one. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

FIG. 8 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on signed sources wherein the accumulator is 2×the input data size.

A first signed source (source 1 801) and a second signed source (source 2 803) each have four packed data elements. Each of these packed data elements stores signed data such as integer data. A third signed source (source 3 809) has two packed data elements, each of which stores signed data. The sizes of the first and second signed sources 801 and 803 are

half that of the third signed source 809. For example, the first and second signed sources 801 and 803 could have 32-bit packed data elements (e.g., single precision floating point) the third signed source 809 could have 64-bit packed data elements (e.g., double precision floating point).

In this illustration, only the two most significant packed data element positions of the first and second signed sources 801 and 803 and the most significant packed data element position of the third signed source 809 are shown. Of course, the other packed data element positions would also be processed.

As illustrated, packed data elements are processed in pairs. For example, the data of the most significant packed data element positions of the first and second signed sources 801 and 803 are multiplied using a multiplier circuit 805, and the data from second most significant packed data element positions of the first and second signed sources 801 and 803 are multiplied using a multiplier circuit 807. In some embodiments, these multiplier circuits 805 and 807 are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source (initial value or previous iteration result) 809. The results of each of the multiplications are added to the signed third source 809 using addition/saturation circuitry 813.

Addition/saturation (accumulator) circuitry 813 preserves a sign of an operand when the addition results in a value that is too big. In particular, saturation evaluation occurs on the infinite precision result between the multi-way-add and the write to the destination or next iteration. When the accumulator 813 is floating point and the input terms are integer, the sum of products and the floating point accumulator input value are turned into infinite precision values (fixed point numbers of hundreds of bits), the addition of the multiplication results and the third input is performed, and a single rounding to the actual accumulator type is performed.

Unsigned saturation means the output values are limited to a maximum unsigned number for that element width (all 1s). Signed saturation means a value is limited to the be in the range between a minimum negative number and a max positive number for that element width (for bytes for example, the range is from −128 ($=-2^7$) to 127($=2^7-1$)).

The result of the addition and saturation check is stored into the signed result 815 in a packed data element position that corresponds to the packed data element position used from the signed third source 809, or passed on to the next iteration if there is one. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

FIG. 9 illustrates an embodiment of a subset of the execution of an iteration of a chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on a signed source and an unsigned source wherein the accumulator is 4×the input data size.

A first signed source (source 1 901) and a second unsigned source (source 2 903) each have four packed data elements. Each of these packed data elements has data such as floating point or integer data. A third signed source (initial value or result 915) has a packed data element of which stores signed data. The sizes of the first and second sources 901 and 903 are a quarter of the third signed source 915. For example, the

first and second sources **901** and **903** could have 16-bit packed data elements (e.g., word) and the third signed source **915** could have 64-bit packed data elements (e.g., double precision floating point or 64-bit integer).

In this illustration, the four most significant packed data element positions of the first and second sources **901** and **903** and the most significant packed data element position of the third signed source **915** are shown. Of course, other packed data element positions would also be processed if there are any.

As illustrated, packed data elements are processed in quadruplets. For example, the data of the most significant packed data element positions of the first and second sources **901** and **903** are multiplied using a multiplier circuit **907**, data from second most significant packed data element positions of the first and second sources **901** and **903** are multiplied using a multiplier circuit **907**, data from third most significant packed data element positions of the first and second sources **901** and **903** are multiplied using a multiplier circuit **909**, and data from the least significant packed data element positions of the first and second sources **901** and **903** are multiplied using a multiplier circuit **911**. In some embodiments, the signed packed data elements of the first source **901** are sign extended and the unsigned packed data elements of the second source **903** are zero extended prior to the multiplications.

In some embodiments, these multiplier circuits **905-911** are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source **915**. The results of each of the multiplications are added using addition circuitry **911**.

The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of the signed source 3 **915** (using a different adder **913** or the same adder **911**).

Finally, the result **919** of the second addition is either stored into the signed destination in a packed data element position that corresponds to the packed data element position used from the signed third source **915**, or passed to the next iteration. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

FIG. **10** illustrates an embodiment of a subset of the execution of an iteration of chained fused multiply accumulate instruction. In particular, this illustrates execution circuitry of an iteration of one packed data element position of the destination. In this embodiment, the chained fused multiply accumulate is operating on a signed source and an unsigned source wherein the accumulator is 4×the input data size.

A first signed source (source 1 **1001**) and a second unsigned source (source 2 **1003**) each have four packed data elements. Each of these packed data elements stores data such as floating point or integer data. A third signed source (initial or previous result **1015**) has a packed data element of which stores signed data. The sizes of the first and second sources **1001** and **1003** are a quarter of the third signed source **1015**. For example, the first and second sources **1001** and **1003** could have 16-bit packed data elements (e.g., word) and the third signed source **1015** could have 64-bit packed data elements (e.g., double precision floating point or 64-bit integer).

In this illustration, the four most significant packed data element positions of the first and second sources **1001** and

1003 and the most significant packed data element position of the third signed source **1015** are shown. Of course, other packed data element positions would also be processed if there are any.

As illustrated, packed data elements are processed in quadruplets. For example, the data of the most significant packed data element positions of the first and second sources **1001** and **1003** are multiplied using a multiplier circuit **1007**, data from second most significant packed data element positions of the first and second sources **1001** and **1003** are multiplied using a multiplier circuit **1007**, data from third most significant packed data element positions of the first and second sources **1001** and **1003** are multiplied using a multiplier circuit **1009**, and data from the least significant packed data element positions of the first and second sources **1001** and **1003** are multiplied using a multiplier circuit **1011**. In some embodiments, the signed packed data elements of the first source **1001** are sign extended and the unsigned packed data elements of the second source **1003** are zero extended prior to the multiplications.

In some embodiments, these multiplier circuits **1005-1011** are reused for other packed data elements positions. In other embodiments, additional multiplier circuits are used so that the packed data elements are processed in parallel. In some contexts, parallel execution is done using lanes that are the size of the signed third source **1015**. The result of the addition of the results of the multiplications is added to the data from most significant packed data element position of the signed source 3 **1015** using addition/saturation circuitry **1013**.

Addition/saturation (accumulator) circuitry **1013** preserves a sign of an operand when the addition results in a value that is too big or too small for signed saturation. In particular, saturation evaluation occurs on the infinite precision result between the multi-way-add and the write to the destination. When the accumulator **1013** is floating point and the input terms are integer, the sum of products and the floating point accumulator input value are turned into infinite precision values (fixed point numbers of hundreds of bits), the addition of the multiplication results and the third input is performed, and a single rounding to the actual accumulator type is performed.

The result **1019** of the addition and saturation check is stored into the signed destination in a packed data element position that corresponds to the packed data element position used from the signed third source **1015**, or passed to the next iteration. In some embodiments, a writemask is applied to this storage such that if a corresponding writemask (bit) is set, the storage happens, and, if not set, the storage does not happen.

FIG. **11** illustrates power-of-two sized SIMD implementations wherein the accumulators use input sizes that are larger than the inputs to the multipliers according to an embodiment. Note the source (to the multipliers) and accumulator values may be signed or unsigned values. For an accumulator having 2×input sizes (in other words, the accumulator input value is twice the size of the packed data element sizes of the sources), table **1101** illustrates different configurations. For byte sized sources, the accumulator uses word or half-precision floating-point (HPFP) values that are 16-bit in size. For word sized sources, the accumulator uses 32-bit integer or single-precision floating-point (SPFP) values that are 32-bit in size. For SPFP or 32-bit integer sized sources, the accumulator uses 64-integer or double-precision floating-point (DPFP) values that are 64-bit in size.

For an accumulator having 4×input sizes (in other words, the accumulator input value is four times the size of the

packed data element sizes of the sources), table **1103** illustrates different configurations. For byte sized sources, the accumulator uses 32-bit integer or single-precision floating-point (SPFP) values that are 32-bit in size. For word sized sources, the accumulator uses 64-bit integer or double-precision floating-point (DPFP) values that are 64-bit in size in some embodiments.

For an accumulator having 8×input sizes (in other words, the accumulator input value is eight times the size of the packed data element sizes of the sources), table **1105** illustrates a configuration. For byte sized sources, the accumulator uses 64-bit integer.

As hinted at earlier, matrix operations circuitry may be included in a core, or as an external accelerator. FIG. **12** illustrates an embodiment of a system utilizing matrix operations circuitry. In this illustration, a plurality of entities are coupled with a ring interconnect **1245**.

A plurality of cores **1201**, **1203**, **1205**, and **1207** provide non-tile based instruction support. In some embodiments, matrix operations circuitry is provided in a core **1203**, and in other embodiments matrix operations circuitry **1211** and **1213** is accessible on the ring interconnect **1245**.

Additionally, one or more memory controllers **1223-1225** are provided to communicate with memory **1233** and **1231** on behalf of the cores and/or matrix operations circuitry.

FIG. **13** illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles. Branch prediction and decode circuitry **1303** performs branch predicting of instructions, decoding of instructions, and/or both from instructions stored in instruction storage **1301**. For example, instructions detailed herein may be stored in instruction storage. In some implementations, separate circuitry is used for branch prediction and in some embodiments, at least some instructions are decoded into one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals using microcode **1305**. The branch prediction and decode circuitry **1303** may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc.

The branch prediction and decode circuitry **1303** is coupled to a rename/allocator circuitry **1307** which is coupled, in some embodiments, to scheduler circuitry **1309**. In some embodiments, these circuits provide register renaming, register allocation, and/or scheduling functionality by performing one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded instruction for execution on execution circuitry out of an instruction pool (e.g., using a reservation station in some embodiments).

The scheduler circuitry **1309** represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) scheduler circuitry **1309** is coupled to, or includes, physical register file(s) **1315**. Each of the physical register file(s) **1315** represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), tiles, etc. In one embodiment, the physical register file(s) **1315** comprises vector registers circuitry, write mask registers circuitry, and scalar registers

circuitry. These register circuits may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) **1315** is overlapped by a retirement circuit **1317** to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement circuit **1317** and the physical register file(s) **1315** are coupled to the execution circuit(s) **1311**.

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor may also include separate instruction and data cache units and a shared L2 cache unit, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

The execution circuitry **1311** a set of one or more execution circuits **1321**, **1323**, and **1327** and a set of one or more memory access circuits **1325**. The execution circuits **1321**, **1323**, and **1327** perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scalar circuitry **1321** performs scalar operations, the vector/SIMD circuitry **1323** performs vector/SIMD operations, and matrix operations circuitry **1327** performs matrix (tile) operations detailed herein.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement a pipeline as follows: 1) an instruction fetch circuit performs fetch and length decoding stages; 2) the branch and decode circuitry **1303** performs a decode stage; 3) the rename/allocator circuitry **1307** performs an allocation stage and renaming stage; 4) the scheduler circuitry **1309** performs a schedule stage; 5) physical register file(s) (coupled to, or included in, the scheduler circuitry **1307** and rename/allocate circuitry **1307** and a memory unit perform a register read/memory read stage; the execution circuitry **1311** performs an execute stage; 6) a memory unit and the physical register file(s) unit(s) perform a write back/memory write stage; 7) various units may be involved in the exception handling stage; and 8) a retirement unit and the physical register file(s) unit(s) perform a commit stage.

The core may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core **1390** includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

FIG. 14 illustrates an embodiment of a processor core pipeline supporting matrix operations using tiles. Branch prediction and decode circuitry 1403 performs branch predicting of instructions, decoding of instructions, and/or both from instructions stored in instruction storage 1401. For example, instructions detailed herein may be stored in instruction storage. In some implementations, separate circuitry is used for branch prediction and in some embodiments, at least some instructions are decoded into one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals using microcode 1405. The branch prediction and decode circuitry 1403 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc.

The branch prediction and decode circuitry 1403 is coupled to a rename/allocator circuitry 1407 which is coupled, in some embodiments, to scheduler circuitry 1409. In some embodiments, these circuits provide register renaming, register allocation, and/or scheduling functionality by performing one or more of: 1) renaming logical operand values to physical operand values (e.g., a register alias table in some embodiments), 2) allocating status bits and flags to the decoded instruction, and 3) scheduling the decoded instruction for execution on execution circuitry out of an instruction pool (e.g., using a reservation station in some embodiments).

The scheduler circuitry 1409 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) scheduler circuitry 1409 is coupled to, or includes, physical register file(s) 1415. Each of the physical register file(s) 1415 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), tiles, etc. In one embodiment, the physical register file(s) 1415 comprises vector registers circuitry, write mask registers circuitry, and scalar registers circuitry. These register circuits may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) 1415 is overlapped by a retirement circuit 1417 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement circuit 1417 and the physical register file(s) 1415 are coupled to the execution circuit(s) 1411.

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor may also include separate instruction and data cache units and a shared L2

cache unit, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

The execution circuitry 1411 a set of one or more execution circuits 1427 and a set of one or more memory access circuits 1425. The execution circuits 1427 perform matrix (tile) operations detailed herein.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement a pipeline as follows: 1) an instruction fetch circuit performs fetch and length decoding stages; 2) the branch and decode circuitry 1403 performs a decode stage; 3) the rename/allocator circuitry 1407 performs an allocation stage and renaming stage; 4) the scheduler circuitry 1409 performs a schedule stage; 5) physical register file(s) (coupled to, or included in, the scheduler circuitry 1407 and rename/allocate circuitry 1407 and a memory unit perform a register read/memory read stage; the execution circuitry 1411 performs an execute stage; 6) a memory unit and the physical register file(s) unit(s) perform a write back/memory write stage; 7) various units may be involved in the exception handling stage; and 8) a retirement unit and the physical register file(s) unit(s) perform a commit stage.

The core may support one or more instructions sets (e.g., the ×86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 1490 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

B. Layout

Throughout this description, data is expressed using row major data layout. Column major users should translate the terms according to their orientation. FIG. 15 illustrates an example of a matrix expressed in row major format and column major format. As shown, matrix A is a 2×3 matrix. When this matrix is stored in row major format, the data elements of a row are consecutive. When this matrix is stored in column major format, the data elements of a column are consecutive. It is a well-known property of matrices that $A^T*B^T=(BA)^T$, where superscript T means transpose. Reading column major data as row major data results in the matrix looking like the transpose matrix.

In some embodiments, row-major semantics are utilized in hardware, and column major data is to swap the operand order with the result being transposes of matrix, but for subsequent column-major reads from memory it is the correct, non-transposed matrix.

For example, if there are two column-major matrices to multiply:

a b	g i k	ag + bh ai + bj ak + bl
c d*	h j l=	cg + dh ci + dj ck + dl
e f		eg + fh ei + fj ek + fl
(3 × 2)	(2 × 3)	(3 × 3)

The input matrices would be stored in linear memory (column-major) as:
a c e b d f
and
g h i j k l.
Reading those matrices as row-major with dimensions 2×3 and 3×2, they would appear as:

a c e	and	g h
b d f		i j
k l		

Swapping the order and matrix multiplying:

g h		a c e	ag + bh cg + dh eg + fh
i j	*	b d f=	ai + bj ci + dj ei + fj
k l			ak + bl ck + dl ek + fl

the transpose matrix is out and can then be stored in in row-major order:

ag + bh	cg + dh	eg + fh	ai + bj	ci + dj	ei + fj	ak + bl	ck + dl	ek + fl

and used in subsequent column major computations, it is the correct un-transposed matrix:

ag + bh	ai + bj	ak + bl
cg + dh	ci + dj	ck + dl
eg + fh	ei + fj	ek + fl

III. Exemplary Usage

FIG. 16 illustrates an example of usage of matrices (tiles). In this example, matrix C 1601 includes two tiles, matrix A 1603 includes one tile, and matrix B 1605 includes two tiles. This figure shows an example of the inner loop of an algorithm to compute a matrix multiplication. In this example, two result tiles, tmm0 and tmm1, from matrix C 1601 are used to accumulate the intermediate results. One tile from the matrix A 1603 (tmm2) is re-used twice as it multiplied by two tiles from the matrix B 1605. Pointers to load a new A tile and two new B tiles from the directions indicated by the arrows. An outer loop, not shown, adjusts the pointers for the C tiles.

The exemplary code as shown includes the usage of a tile configuration instruction and is executed to configure tile usage, load tiles, a loop to process the tiles, store tiles to memory, and release tile usage.

FIG. 17 illustrates an embodiment of usage of matrices (tiles). At 1701, tile usage is configured. For example, a TILECONFIG instruction is executed to configure tile usage including setting a number of rows and columns per tile. Typically, at least one matrix (tile) is loaded from memory

at 1703. At least one matrix (tile) operation is performed at 1705 using the matrices (tiles). At 1707, at least one matrix (tile) is stored out to memory and a context switch can occur at 1709.

IV. Exemplary Instruction

A. Store Tile Configuration

As discussed above, tile usage typically needs to be configured prior to use. For example, full usage of all rows and columns may not be needed. Not only does not configuring these rows and columns save power in some embodiments, but the configuration may be used to determine if an operation will generate an error. For example, a matrix multiplication of the form (N×M)*(L*N) will typically not work if M and L are not the same.

Detailed herein are embodiments of a store matrix (tile) configuration ("STTILECFG") instruction and its execution. Prior to using matrices using tiles, in some embodiments, tile support is to be configured (typically using a TILECON-FIG instruction). For example, how many rows and columns per tile, tiles that are to be used, etc. are configured. A STTILECFG instruction stores that configuration into memory. In particular, an execution of the STTILECFG instruction causes a matrix (tile) configuration to be stored to memory for later use to configure matrix (tile) usage.

i. Exemplary Execution

FIG. 18 illustrates an exemplary execution of a store tile configuration (STTILECFG) instruction. The STTILECFG instruction format includes fields for an opcode and a destination memory address.

As illustrated, the STTILECFG instruction uses the address as a pointer to a memory 1801 location to contain the description of the matrices (tiles) to be supported 1803.

Execution circuitry 1811 of a processor/core 1805 performs the STTILECFG by retrieving aspects of a tile description 1803 from tile configurations 1817 and storing into memory 1801 via a memory controller 1815. The tile configurations 1817 detail what tiles for a palette are configured (the number of rows and columns in each tile) and a marking that matrix support is in use. In particular, instruction execution resources 1811 are configured to use tiles as specified by the tile configuration 1817. The instruction execution resources may also include a machine specific register or configuration register to indicate tile usage. Additional values such as in-use and start values are also set. The tile configurations 1817 utilize one or more registers 1819 to store tile usage and configuration information.

ii. Exemplary Stored Matrix (Tile) Description

FIG. 19 illustrates an embodiment of a description of the matrices (tiles) to be supported. This is the description that is to be stored upon an execution of a STTILECFG instruction. In this example, each field is a byte. In byte[0], a palette ID 1901 is stored. The palette ID is used to index a palette table 1813 which stores, per palette ID, a number of bytes in a tile, and bytes per row of the tiles that are associated with this ID as defined by the configuration.

Byte 1 stores a value to be stored in a "startRow" register 1903 and byte 2 stores a value to be stored in a "startP" register 1905. To support restarting instructions after break events, the instructions store information these registers. To support restarting instructions after these events, the instructions store information in these registers. The startRow value indicates the row that should be used for restart. The startP value indicates the position within the row for store operations used in pairs and, in some embodiments, indicates the lower half of the row (in the lower tile of a pair) or the higher

half of the row (in the higher tile of the pair). Generally, the position in the row (the column) is not needed.

With the exception of TILECONFIG and STTILECFG, successfully executing matrix (tile) instructions will set both startRow and startP to zero.

Any time an interrupted matrix (tile) instruction is not restarted, it is the responsibility of software to zero the startRow and startP values. For example, unmasked floating point exception handlers might decide to finish the operation in software and change the program counter value to another instruction, usually the next instruction. In this case the software exception handler must zero the startRow and startP values in the exception presented to it by the operating system before resuming the program. The operating system will subsequently reload those values using a restore instruction.

Byte 3 stores an indication of pairs (1 b per tile) of tiles **1907**.

Bytes 16-17 store the number of rows **1913** and columns **1915** for tile 0, bytes 18-19 store the number of rows and columns for tile 1, etc. In other words, each 2 byte group specifies a number of rows and columns for a tile. If a group of 2 bytes is not used to specify tile parameters, they should have the value zero. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a fault. Unconfigured tiles are set to an initial state with 0 rows, 0 columns.

Finally, the configuration in memory typically ends with an ending delineation such as all zeros for several consecutive bytes.

iii. Exemplary Tile Storage

FIGS. **20**(A)-(D) illustrate examples of register(s) **1819**. FIG. **20**(A) illustrates a plurality of registers **1819**. As shown each tile (TMM0 **2001** . . . TMMN **2003**) has a separate register with each register storing a row and column size for that particular tile. StartP and StartRow are stored in separate registers **2011** and **2013**. One or more status registers **2015** are set (e.g., TILES_CONFIGURED=1) to indicate tiles are configured for use.

FIG. **20**(B) illustrates a plurality of registers **1819**. As shown each tile has separate registers for its rows and columns. For example, TMM0 rows configuration **2021**, TMM0 columns configuration **2023**, StartP and StartRow are stored in separate registers **2011** and **2013**. One or more status registers **2015** are set (e.g., TILES_CONFIG-URED=1) to indicate tiles are configured for use.

FIG. **20**(C) illustrates a single register **1819**. As shown, this register stores tile configurations (rows and columns per tile) **2031**, StartP **2011**, and StartRow **2013** are stored in single register as packed data registers. One or more status registers **2015** are set (e.g., TILES_CONFIGURED=1) to indicate tiles are configured for use.

FIG. **20**(D) illustrates a plurality of registers **1819**. As shown, a single register stores tile configurations (rows and columns per tile) **2031**. StartP and StartRow are stored in separate registers **2011** and **2013**. One or more status registers **2015** are set (e.g., TILES_CONFIGURED=1) to indicate tiles are configured for use.

Other combinations are contemplated such as combining the start registers into a single register where they are shown separately, etc.

iv. Exemplary Format(s)

An embodiment of a format for a STTILECFG instruction is STTILECFG Address. In some embodiments, STTILECFG is the opcode mnemonic of the instruction. Address

is a pointer to a matrix (tile) description in memory. In some embodiments, the address field is a R/M value (such as **2446**).

In embodiments, encodings of the instruction include a scale-index-base (SIB) type memory addressing operand that indirectly identifies multiple indexed destination locations in memory (e.g., field **2450**). In one embodiment, an SIB type memory operand may include an encoding identifying a base address register. The contents of the base address register may represent a base address in memory from which the addresses of the particular destination locations in memory are calculated. For example, the base address may be the address of the first location in a block of potential destination locations for an extended vector instruction. In one embodiment, an SIB type memory operand may include an encoding identifying an index register. Each element of the index register may specify an index or offset value usable to compute, from the base address, an address of a respective destination location within a block of potential destination locations. In one embodiment, an SIB type memory operand may include an encoding specifying a scaling factor to be applied to each index value when computing a respective destination address. For example, if a scaling factor value of four is encoded in the SIB type memory operand, each index value obtained from an element of the index register may be multiplied by four and then added to the base address to compute a destination address.

In one embodiment, an SIB type memory operand of the form vm32{x,y,z} may identify a vector array of memory operands specified using SIB type memory addressing. In this example, the array of memory addresses is specified using a common base register, a constant scaling factor, and a vector index register containing individual elements, each of which is a 32-bit index value. The vector index register may be a 128-bit register (e.g., XMM) register (vm32x), a 256-bit (e.g., YMM) register (vm32y), or a 512-bit (e.g., ZMM) register (vm32z). In another embodiment, an SIB type memory operand of the form vm64{x,y,z} may identify a vector array of memory operands specified using SIB type memory addressing. In this example, the array of memory addresses is specified using a common base register, a constant scaling factor, and a vector index register containing individual elements, each of which is a 64-bit index value. The vector index register may be a 128-bit register (e.g., XMM) register (vm64x), a 256-bit (e.g., YMM) register (vm64y) or a 512-bit (e.g., ZMM) register (vm64z).

v. Exemplary Method(s) of Execution

FIG. **21** illustrates an embodiment of method performed by a processor to process a STTILECFG instruction.

At **2101**, an instruction is fetched. For example, a STTILECFG instruction is fetched. An embodiment of the STTILECFG instruction includes fields for an opcode (store tile configuration) and a memory address.

The fetched instruction is decoded at **2103**. For example, the fetched STTILECFG instruction is decoded by decode circuitry such as that detailed herein.

A description found at the memory address of the memory address operand is retrieved at **2105** and the decoded instruction is scheduled (as needed).

At **2107**, the decoded instruction is executed by execution circuitry (hardware) such as that detailed herein. For the STTILECFG instruction, the execution will cause execution circuitry to store configured matrix (tile) information (for example, the number of rows and columns, etc. detailed above) as a matrix (tile) usage description at the memory

address. For example, the configuration stored in one or more registers **1819** is converted into a matrix (tile) description.

In some embodiments, the instruction is committed or retired at **2109**.

FIG. **22** illustrates a more detailed description of an execution of a STTILECFG instruction using memory addressing. Typically, this is performed by execution circuitry such as that detailed above.

In some embodiments, a check is first performed to determine whether matrix (tile) usage is configured at **2201**. For example, has a matrix (tile) usage bit been set.

When matrix (tile) usage has not been configured, then all zeros are written to the memory address at **2203**. When matrix (tile) usage has been configured, a matrix (tile) description is created and written into memory at the memory address at **2205**.

The creation of the matrix (tile) description (which may be done before writing or done during the writing) includes many different items being read and stored.

Several items are not dependent upon particular matrices (tiles) being configured for use and are typically made a part of the description first at **2207**. For example, the palette ID is the first byte of the matrix (tile) description, the startRow value is stored into a second, consecutive byte, the startP value is stored into a third, consecutive byte, and pair indicators are stored into a fourth, consecutive byte of the description. For example, in byte[0], the palette ID **1901** is stored. Byte 1 stores a value from the "startRow" register **1903** and byte 2 stores a value for a "startP" register **1905**. Byte 2 stores the startP value indicates the position within the row for store operations. Byte 3 stores an indication of pairs (1 b per tile) of tiles **1907**.

Next the matrix (tile) specific information is made a part of the description at **2209** through **2215**. For example, at **2209**, row and column configuration information about a matrix (tile) (e.g., tile 0) is added to the description. Typically, this is stored in bytes 16-17. Bytes 4-15 are set to zero in some embodiments, however, not all embodiments use this convention and in some embodiments, the storage of matrix (tile) specific data begins at byte 4 of the description.

A determination of whether all configuration (row/column) information for the matrices (tiles) have been added to the description is made at **2211**. If not, then the matrix (tile) number is increased at **2213** and that matrix (tile) information is added at **2209**. If so, then the adding of matrix (tile) specific information is complete at **2215**. In some embodiments, the remainder of the description is zeroed at this point.

vi. Exemplary Pseudocode

FIG. **23** illustrates exemplary pseudocode for an execution of a STTILECFG instruction.

vii. Examples

Example 1. An apparatus comprising: decode circuitry to decode an instruction having an opcode and a memory location; and execution circuitry to execute the decoded instruction to retrieve configuration information about usage of storage for two-dimensional data structures and store the retrieved configuration information as description data at the memory location.

Example 2. The apparatus of example 1, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

Example 3. The apparatus of example 1, wherein the storage is a plurality of packed data registers and memory, and the two-dimensional data structures are overlaid on the plurality of packed data registers and memory.

Example 4. The apparatus of any of examples 1-3, wherein the memory location is stored in a scale-index-base format.

Example 5. The apparatus of any of examples 1-4, wherein the description data comprises: 1) an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure; 2) restart information used in two-dimensional data structure operations; and 3) indications of a number of rows and columns per two-dimensional data structure.

Example 6. The apparatus of example 5, wherein the description data is to further comprise an indication of pairs two-dimensional data structures.

Example 7. The apparatus of any of examples 1-6, wherein the description data is retrieved from at least one register of the apparatus.

Example 8. A method comprising: decoding an instruction having an opcode and a memory location; and executing the decoded instruction to retrieve configuration information about usage of storage for two-dimensional data structures and store the retrieved configuration information as description data at the memory location.

Example 9. The method of example 8, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

Example 10. The method of example 8, wherein the storage is a plurality of packed data registers and memory, and the two-dimensional data structures are overlaid on the plurality of packed data registers and memory.

Example 11. The method of any of examples 8-10, wherein the memory location is stored in a scale-index-base format.

Example 12. The method of any of examples 8-11, wherein the description data comprises: 1) an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure; 2) restart information used in two-dimensional data structure operations; and 3) indications of a number of rows and columns per two-dimensional data structure.

Example 13. The method of example 12, wherein the description data is to further comprise an indication of pairs two-dimensional data structures.

Example 14. The method of any of examples 8-13, wherein the description data is retrieved from at least one register of the apparatus.

Example 15. A non-transitory machine readable medium storing an occurrence of an instruction, wherein a processor upon encountering the occurrence of the instruction is to perform a method comprising: decoding the instruction having an opcode and a memory location; and executing the decoded instruction to retrieve configuration information about usage of storage for two-dimensional data structures and store the retrieved configuration information as description data at the memory location.

Example 16. The non-transitory machine readable medium of example 15, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

Example 17. The non-transitory machine readable medium of example 15, wherein the storage is a plurality of

packed data registers and memory, and the two-dimensional data structures are overlaid on the plurality of packed data registers and memory.

Example 18. The non-transitory machine readable medium of any of examples 15-17, wherein the memory location is stored in a scale-index-base format.

Example 19. The non-transitory machine readable medium of any of examples 15-18, wherein the description data comprises: 1) an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure; 2) restart information used in two-dimensional data structure operations; and 3) indications of a number of rows and columns per two-dimensional data structure.

Example 20. The non-transitory machine readable medium of example 19, wherein the description data is to further comprise an indication of pairs two-dimensional data structures.

Example 21. The non-transitory machine readable medium of any of examples 15-20, wherein the description data is retrieved from at least one register of the apparatus.

IV. Detailed Exemplary Systems, Processors, and Emulation

Detailed herein are examples of hardware, software, etc. to execute the above described instructions. For example, what is described below details aspects of instruction execution including various pipeline stages such as fetch, decode, schedule, execute, retire, etc.

An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed (opcode) and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down though the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

A. Exemplary Instruction Formats

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

VEX Instruction Format

VEX encoding allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 248 bits. The use of a VEX prefix provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as A=A+B, which overwrites a source operand. The use of a VEX prefix enables operands to perform nondestructive operations such as A=B+C.

FIG. 24A illustrates an exemplary instruction format including a VEX prefix 2402, real opcode field 2430, Mod R/M byte 2440, SIB byte 2450, displacement field 2462, and IMM8 2472. FIG. 24B illustrates which fields from FIG. 24A make up a full opcode field 2474 and a base operation field 2441. FIG. 24C illustrates which fields from FIG. 24A make up a register index field 2444.

VEX Prefix (Bytes 0-2) 2402 is encoded in a three-byte form. The first byte is the Format Field 2490 (VEX Byte 0, bits [7:0]), which contains an explicit C4 byte value (the unique value used for distinguishing the C4 instruction format). The second-third bytes (VEX Bytes 1-2) include a number of bit fields providing specific capability. Specifically, REX field 2405 (VEX Byte 1, bits [7-5]) consists of a VEX.R bit field (VEX Byte 1, bit [7]-R), VEX.X bit field (VEX byte 1, bit [6]-X), and VEX.B bit field (VEX byte 1, bit[5]-B). Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding VEX.R, VEX.X, and VEX.B. Opcode map field 2415 (VEX byte 1, bits [4:0]-mmmmm) includes content to encode an implied leading opcode byte. W Field 2464 (VEX byte 2, bit [7]-W)—is represented by the notation VEX.W, and provides different functions depending on the instruction. The role of VEX.vvvv 2420 (VEX Byte 2, bits [6:3]-vvvv) may include the following: 1) VEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L 2468 Size field (VEX byte 2, bit [2]-L)=0, it indicates 248 bit vector; if VEX.L=1, it indicates 256 bit vector. Prefix encoding field 2425 (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field 2441.

Real Opcode Field 2430 (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field 2440 (Byte 4) includes MOD field 2442 (bits [7-6]), Reg field 2444 (bits [5-3]), and R/M field 2446 (bits [2-0]). The role of Reg field 2444 may include the following: encoding either the destination register operand or a source register operand (the rrr of Rrrr), or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 2446 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB)—The content of Scale field 2450 (Byte 5) includes SS2452 (bits [7-6]), which is used for memory address generation. The contents of SIB.xxx 2454 (bits [5-3]) and SIB.bbb 2456 (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb.

The Displacement Field 2462 and the immediate field (IMM8) 2472 contain data.

B. Exemplary Register Architecture

FIG. 25 is a block diagram of a register architecture 2500 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 2510 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 28 zmm registers are overlaid on registers ymm0-15.

The lower order 128 bits of the lower 28 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15.

General-purpose registers **2525**—in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing ×86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (×87 stack) **2545**, on which is aliased the MMX packed integer flat register file **2550**—in the embodiment illustrated, the ×87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the ×87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

In some embodiments, tiles **2510** are supported using an overlay over physical registers. For example, a tile may utilize 16 1,024-bit registers, 32 512-bit registers, etc. depending on the implementation.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Exemplary Core Architectures, Processors, and Computer Architectures

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures. Detailed herein are circuits (units) that comprise exemplary cores, processors, etc.

C. Exemplary Core Architectures

In-Order and Out-of-Order Core Block Diagram

FIG. **26**A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. FIG. **26**B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a

processor according to embodiments of the invention. The solid lined boxes in FIGS. **26**A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In FIG. **26**A, a processor pipeline **2600** includes a fetch stage **2602**, a length decode stage **2604**, a decode stage **2606**, an allocation stage **2608**, a renaming stage **2610**, a scheduling (also known as a dispatch or issue) stage **2612**, a register read/memory read stage **2614**, an execute stage **2616**, a write back/memory write stage **2618**, an exception handling stage **2622**, and a commit stage **2624**.

FIG. **26**B shows processor core **2690** including a front end unit **2630** coupled to an execution engine unit **2650**, and both are coupled to a memory unit **2670**. The core **2690** may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core **2690** may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit **2630** includes a branch prediction unit **2632** coupled to an instruction cache unit **2634**, which is coupled to an instruction translation lookaside buffer (TLB) **2636**, which is coupled to an instruction fetch unit **2638**, which is coupled to a decode unit **2640**. The decode unit **2640** (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit **2640** may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core **2690** includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit **2640** or otherwise within the front end unit **2630**). The decode unit **2640** is coupled to a rename/allocator unit **2652** in the execution engine unit **2650**.

The execution engine unit **2650** includes the rename/allocator unit **2652** coupled to a retirement unit **2654** and a set of one or more scheduler unit(s) **2656**. The scheduler unit(s) **2656** represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) **2656** is coupled to the physical register file(s) unit(s) **2658**. Each of the physical register file(s) units **2658** represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit **2658** comprises a vector registers unit and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) **2658** is overlapped by the retirement unit **2654** to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and

a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit **2654** and the physical register file(s) unit(s) **2658** are coupled to the execution cluster(s) **2660**. The execution cluster(s) **2660** includes a set of one or more execution units **2662** and a set of one or more memory access units **2664**. The execution units **2662** may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) **2656**, physical register file(s) unit(s) **2658**, and execution cluster(s) **2660** are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/ vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster— and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) **2664**). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units **2664** is coupled to the memory unit **2670**, which includes a data TLB unit **2672** coupled to a data cache unit **2674** coupled to a level 2 (L2) cache unit **2676**. In one exemplary embodiment, the memory access units **2664** may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit **2672** in the memory unit **2670**. The instruction cache unit **2634** is further coupled to a level 2 (L2) cache unit **2676** in the memory unit **2670**. The L2 cache unit **2676** is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline **2600** as follows: 1) the instruction fetch **2638** performs the fetch and length decoding stages **2602** and **2604**; 2) the decode unit **2640** performs the decode stage **2606**; 3) the rename/allocator unit **2652** performs the allocation stage **2608** and renaming stage **2610**; 4) the scheduler unit(s) **2656** performs the schedule stage **2612**; 5) the physical register file(s) unit(s) **2658** and the memory unit **2670** perform the register read/memory read stage **2614**; the execution cluster **2660** perform the execute stage **2616**; 6) the memory unit **2670** and the physical register file(s) unit(s) **2658** perform the write back/memory write stage **2618**; 7) various units may be involved in the exception handling stage **2622**; and 8) the retirement unit **2654** and the physical register file(s) unit(s) **2658** perform the commit stage **2624**.

The core **2690** may support one or more instructions sets (e.g., the ×86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core **2690** includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units **2634/2674** and a shared L2 cache unit **2676**, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Specific Exemplary In-Order Core Architecture

FIGS. **27A-B** illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

FIG. **27A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network **2702** and with its local subset of the Level 2 (L2) cache **2704**, according to embodiments of the invention. In one embodiment, an instruction decoder **2700** supports the ×86 instruction set with a packed data instruction set extension. An L1 cache **2706** allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit **2708** and a vector unit **2710** use separate register sets (respectively, scalar registers **2712** and vector registers **2714**) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache **2706**, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

The local subset of the L2 cache **2704** is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache **2704**. Data read by a processor core is stored in its L2 cache subset **2704** and can be accessed quickly, in parallel with other processor cores accessing their own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset **2704** and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1024-bits wide per direction in some embodiments.

FIG. **27B** is an expanded view of part of the processor core in FIG. **27A** according to embodiments of the invention. FIG. **27B** includes an L1 data cache **2706A** part of the L1 cache **2704**, as well as more detail regarding the vector unit **2710** and the vector registers **2714**. Specifically, the

vector unit **2710** is a 28-wide vector processing unit (VPU) (see the 16-wide ALU **2728**), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit **2720**, numeric conversion with numeric convert units **2722A-B**, and replication with replication unit **2724** on the memory input.

Processor With Integrated Memory Controller and Graphics

FIG. **28** is a block diagram of a processor **2800** that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in FIG. **28** illustrate a processor **2800** with a single core **2802A**, a system agent **2810**, a set of one or more bus controller units **2816**, while the optional addition of the dashed lined boxes illustrates an alternative processor **2800** with multiple cores **2802A-N**, a set of one or more integrated memory controller unit(s) **2814** in the system agent unit **2810**, and special purpose logic **2808**.

Thus, different implementations of the processor **2800** may include: 1) a CPU with the special purpose logic **2808** being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores **2802A-N** being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores **2802A-N** being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores **2802A-N** being a large number of general purpose in-order cores. Thus, the processor **2800** may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many inte-grated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor **2800** may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

The memory hierarchy includes one or more levels of cache within the cores **2804A-N**, a set or one or more shared cache units **2806**, and external memory (not shown) coupled to the set of integrated memory controller units **2814**. The set of shared cache units **2806** may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit **2812** interconnects the inte-grated graphics logic **2808**, the set of shared cache units **2806**, and the system agent unit **2810**/integrated memory controller unit(s) **2814**, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units **2806** and cores **2802-A-N**.

In some embodiments, one or more of the cores **2802A-N** are capable of multi-threading. The system agent **2810** includes those components coordinating and operating cores **2802A-N**. The system agent unit **2810** may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores **2802A-N** and the integrated graphics logic **2808**. The display unit is for driving one or more externally connected displays.

The cores **2802A-N** may be homogenous or heteroge-neous in terms of architecture instruction set; that is, two or more of the cores **2802A-N** may be capable of execution the same instruction set, while others may be capable of execut-ing only a subset of that instruction set or a different instruction set.

D. Exemplary Computer Architectures

FIGS. **29-32** are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, serv-ers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro control-lers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to FIG. **29**, shown is a block diagram of a system **2900** in accordance with one embodiment of the present invention. The system **2900** may include one or more processors **2910**, **2915**, which are coupled to a con-troller hub **2920**. In one embodiment the controller hub **2920** includes a graphics memory controller hub (GMCH) **2990** and an Input/Output Hub (IOH) **2950** (which may be on separate chips); the GMCH **2990** includes memory and graphics controllers to which are coupled memory **2940** and a coprocessor **2945**; the IOH **2950** is couples input/output (I/O) devices **2960** to the GMCH **2990**. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory **2940** and the coprocessor **2945** are coupled directly to the pro-cessor **2910**, and the controller hub **2920** in a single chip with the IOH **2950**.

The optional nature of additional processors **2915** is denoted in FIG. **29** with broken lines. Each processor **2910**, **2915** may include one or more of the processing cores described herein and may be some version of the processor **2800**.

The memory **2940** may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub **2920** communicates with the processor(s) **2910**, **2915** via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface, or similar connection **2995**.

In one embodiment, the coprocessor **2945** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embed-ded processor, or the like. In one embodiment, controller hub **2920** may include an integrated graphics accelerator.

There can be a variety of differences between the physical resources **2910**, **29155** in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

In one embodiment, the processor **2910** executes instruc-tions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor **2910** recognizes these coproces-sor instructions as being of a type that should be executed by the attached coprocessor **2945**. Accordingly, the processor **2910** issues these coprocessor instructions (or control sig-nals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor **2945**. Coprocessor (s) **2945** accept and execute the received coprocessor instructions.

Referring now to FIG. **30**, shown is a block diagram of a first more specific exemplary system **3000** in accordance

with an embodiment of the present invention. As shown in FIG. **30**, multiprocessor system **3000** is a point-to-point interconnect system, and includes a first processor **3070** and a second processor **3080** coupled via a point-to-point interconnect **3050**. Each of processors **3070** and **3080** may be some version of the processor **2800**. In one embodiment of the invention, processors **3070** and **3080** are respectively processors **2910** and **2915**, while coprocessor **3038** is coprocessor **2945**. In another embodiment, processors **3070** and **3080** are respectively processor **2910** coprocessor **2945**.

Processors **3070** and **3080** are shown including integrated memory controller (IMC) units **3072** and **3082**, respectively. Processor **3070** also includes as part of its bus controller units point-to-point (P-P) interfaces **3076** and **3078**; similarly, second processor **3080** includes P-P interfaces **3086** and **3088**. Processors **3070**, **3080** may exchange information via a point-to-point (P-P) interface **3050** using P-P interface circuits **3078**, **3088**. As shown in FIG. **30**, IMCs **3072** and **3082** couple the processors to respective memories, namely a memory **3032** and a memory **3034**, which may be portions of main memory locally attached to the respective processors.

Processors **3070**, **3080** may each exchange information with a chipset **3090** via individual P-P interfaces **3052**, **3054** using point to point interface circuits **3076**, **3094**, **3086**, **3098**. Chipset **3090** may optionally exchange information with the coprocessor **3038** via a high-performance interface **3092**. In one embodiment, the coprocessor **3038** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

Chipset **3090** may be coupled to a first bus **3016** via an interface **3096**. In one embodiment, first bus **3016** may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another I/O interconnect bus, although the scope of the present invention is not so limited.

As shown in FIG. **30**, various I/O devices **3014** may be coupled to first bus **3016**, along with a bus bridge **3018** which couples first bus **3016** to a second bus **3020**. In one embodiment, one or more additional processor(s) **3015**, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus **3016**. In one embodiment, second bus **3020** may be a low pin count (LPC) bus. Various devices may be coupled to a second bus **3020** including, for example, a keyboard and/or mouse **3022**, communication devices **3027** and a storage unit **3028** such as a disk drive or other mass storage device which may include instructions/code and data **3030**, in one embodiment. Further, an audio I/O **3024** may be coupled to the second bus **3016**. Note that other architectures are possible. For example, instead of the point-to-point architecture of FIG. **30**, a system may implement a multi-drop bus or other such architecture.

Referring now to FIG. **31**, shown is a block diagram of a second more specific exemplary system **3100** in accordance with an embodiment of the present invention. Like elements in FIGS. **30** and **31** bear like reference numerals, and certain aspects of FIG. **30** have been omitted from FIG. **31** in order to avoid obscuring other aspects of FIG. **31**.

FIG. **31** illustrates that the processors **3070**, **3080** may include integrated memory and I/O control logic ("CL") **3172** and **3182**, respectively. Thus, the CL **3172**, **3182** include integrated memory controller units and include I/O control logic. FIG. **31** illustrates that not only are the memories **3032**, **3034** coupled to the CL **3172**, **3182**, but also that I/O devices **3114** are also coupled to the control logic **3072**, **3082**. Legacy I/O devices **3115** are coupled to the chipset **3090**.

Referring now to FIG. **32**, shown is a block diagram of a SoC **3200** in accordance with an embodiment of the present invention. Similar elements in FIG. **28** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In FIG. **32**, an interconnect unit(s) **3202** is coupled to: an application processor **3210** which includes a set of one or more cores **322A-N**, cache units **2804A-N**, and shared cache unit(s) **2806**; a system agent unit **2810**; a bus controller unit(s) **2816**; an integrated memory controller unit(s) **2814**; a set or one or more coprocessors **3220** which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit **3230**; a direct memory access (DMA) unit **3232**; and a display unit **3240** for coupling to one or more external displays. In one embodiment, the coprocessor(s) **3220** include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

Program code, such as code **3030** illustrated in FIG. **30**, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as "IP cores" may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device,

including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

E. Emulation (Including Binary Translation, Code Morphing, etc.)

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

FIG. **33** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIG. **33** shows a program in a high level language **3302** may be compiled using an first compiler **3304** to generate a first binary code (e.g., x86) **3306** that may be natively executed by a processor with at least one first instruction set core **3316**. In some embodiments, the processor with at least one first instruction set core **3316** represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one '86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The first compiler **3304** represents a compiler that is operable to generate binary code of the first instruction set **3306** (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one first instruction set core **3316**. Similarly, FIG. **33** shows the program in the high level language **3302** may be compiled using an alternative instruction set compiler **3308** to generate alternative instruction set binary code **3310** that may be natively executed by a processor without at least one first instruction set core **3314** (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of

Sunnyvale, CA). The instruction converter **3312** is used to convert the first binary code **3306** into code that may be natively executed by the processor without an first instruction set core **3314**. This converted code is not likely to be the same as the alternative instruction set binary code **3310** because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter **3312** represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have a first instruction set processor or core to execute the first binary code **3306**.

We claim:

1. An apparatus comprising:

decode circuitry configured to decode at least an instance of a single instruction having an opcode and memory location information, wherein the opcode is to indicate execution circuitry is to retrieve configuration information about how storage of the apparatus was configured as two-dimensional data structures to be used by execution circuitry configured to perform two-dimensional matrix operations and store the retrieved configuration information as description data at a memory location to be identified from the memory location information, wherein the description data is to include a palette identifier, and a value to indicate a row to be used to restart; and

execution circuitry, in response to the decoded instance of the single instruction, to be configured to execute the decoded instance of the single instruction according to the opcode.

2. The apparatus of claim **1**, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

3. The apparatus of claim **1**, wherein the memory location is to be accessed using information to be stored in a scale-index-base format.

4. The apparatus of claim **1**, wherein the description data comprises an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure and indications of a number of rows and columns per two-dimensional data structure.

5. The apparatus of claim **1**, wherein the description data is to be retrieved from at least one register of the apparatus.

6. An apparatus comprising:

a processor core comprising decode circuitry configured to decode at least an instance of a single instruction having an opcode and memory location information, wherein the opcode is to indicate execution circuitry is to retrieve configuration information about how storage of the apparatus was configured as two-dimensional data structures to be used by execution circuitry configured to perform two-dimensional matrix operations and store the retrieved configuration information as description data at a memory location to be identified from the memory location information, wherein the description data is to include a palette identifier, and a value to indicate a row to be used to restart; and

an accelerator comprising execution circuitry, in response to the decoded instance of the single instruction, to be configured to execute the decoded instance of the single instruction according to the opcode.

7. The apparatus of claim 6, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

8. The apparatus of claim 6, wherein the memory location is to be accessed using information to be stored in a scale-index-base format.

9. The apparatus of claim 6, wherein the description data comprises an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure and indications of a number of rows and columns per two-dimensional data structure.

10. The apparatus of claim 6, wherein the description data is to be retrieved from at least one register of the apparatus.

11. A non-transitory machine readable medium storing an occurrence of an instruction, wherein a processor upon encountering the occurrence of the instruction is to perform a method comprising:

decoding at least an instance of a single instruction having an opcode and memory location information, wherein the opcode is to indicate execution circuitry is to retrieve configuration information about how storage of an apparatus was configured as two-dimensional data structures to be used by execution circuitry configured to perform two-dimensional matrix operations and store the retrieved configuration information as description data at a memory location to be identified

from the memory location information, wherein the description data is to include a palette identifier, and a value to indicate a row to be used to restart; and

executing the decoded instance of the single instruction to retrieve configuration information about how storage of an apparatus was configured as two-dimensional data structures to be used by execution circuitry configured to perform two-dimensional matrix operations and store the retrieved configuration information as description data at a memory location identified from the memory location information.

12. The non-transitory machine readable medium of claim 11, wherein the storage is a plurality of packed data registers and the two-dimensional data structures are overlaid on the plurality of packed data registers.

13. The non-transitory machine readable medium of claim 11, wherein the memory location is accessed using information to be stored in a scale-index-base format.

14. The non-transitory machine readable medium of claim 11, wherein the description data comprises an index into a table which is to store a number of bytes in a two-dimensional data structure, and bytes per row of the two-dimensional data structure and indications of a number of rows and columns per two-dimensional data structure.

15. The non-transitory machine readable medium of claim 11, wherein the description data is to be retrieved from at least one register.

* * * * *